来自中国顶尖通信企业的资深工程师多年驱动开发经验的梳理
系统剖析Android底层开发，填补了底层开发书籍的空白
紧扣驱动开发的本质，一层一层向读者展现设备驱动开发的核心技术
驱动开发工程性极强，案例翔实，代码丰富，引导读者边阅读边实践

嵌入式Linux

Driver

Android 驱动开发权威指南

杨柳 编著

机械工业出版社
China Machine Press

Android驱动开发权威指南

杨柳　著

ISBN：978-7-111-45182-2

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.bbbvip.com

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

前言

目前关于Android开发的书籍比较多，但大多数或立足于Android的上层，讲解如何开发Android应用，或偏重于Android Framework的繁杂机制，而缺乏讲述Android驱动及具体实践的书籍。由于工作的原因，本人正在根据多年Linux与Android驱动开发的经验，整理相关的新员工培训材料。为此，应张国强先生相邀，把相关资料整理成册，以飨读者，希望能让读者对Android有更深、更全面的理解，期待对从事Android驱动开发的工程人员的实际工作有所帮助。

我们知道Android底层是基于Linux内核的，因此要基于Android开发智能手机或终端，就少不了Linux内核与底层驱动的开发。据了解，这方面的开发人员目前还很缺乏，其中一个重要的原因就是Linux是一个与Windows一样复杂的软件系统，要理解它，本身不是一件容易的事，因此许多感兴趣的读者在翻阅相关书籍之后就知难而退了。根据个人的工作经验，Linux内核和底层驱动的开发其实并没有想象中的那么难，大概是相关书籍首先就讲进程调度、信号同步等复杂技术，让读者立刻坠入云里雾里，特别是若没弄明白设备驱动的本质，当真正讲到相关驱动开发技术时，心气已去了大半，更不用谈利用所学的知识进行实践了。

因此，我们在讲述相关驱动开发基础知识前，先让大家初步了解Linux内核，再着手讲设备输入/输出（I/O），接着由设备I/O引入DMA与中断处理，再讲驱动的并发处理技术，最后才讲设备驱动向上层提供的同步与异步访问方法。以设备I/O本质为源，根据前后因果，一层一层向读者展现设备驱动开发的核心技术。只有这样，我们在编写或修改相应的Android驱动时，采用相关的技术才会得心应手，所创建的驱动才能健壮地与原有驱动与系统融为一体。而且经过这样一个学习与实践过程，相信读者对于基于Linux内核的Android会有更清楚的认识。

Android驱动开发是一门工程性很强的技术，因此本书从实际开发需要出发，同时反复强调动手实践。另外，我们认为一个合格的Android驱动程序开发工程师，除了要会编写驱动程序，向上层程序提供相应设备模块功能外，还应会编写DVT（Design Verify Test）测试程序，以验证所创建的驱动程序。因此在第三篇“Android驱动实践篇”中，除

了讲解Android驱动开发原理，还讲解了Android HAL等与Android驱动紧密相关的中间层知识；在讲述具体Android设备子系统时，从该子系统整体工作机制出发，力求让读者理解开发的驱动如何成为相关子系统的有机部分，为Android驱动开发人员提供驱动开发以及DVT测试程序开发指导。

通过本书的学习，读者将对Android整体有更深且更全面的认识。至于Android APK等程序开发的具体知识，相关的书籍与互联网上有很多资料，读者可以在进行具体DVT之类开发时，查阅相关技术的资料。

众所周知，驱动开发是与操作系统紧密相关的，因此Android驱动开发的工作应该属于系统级工程师的任务，要求我们比Android应用等开发人员知识面更广，这一点在本人编写本书时感触特别深。站在许多前辈的肩膀上，糅合了自己的经验与理解，历时8个月，终于成稿，心中依然忐忑。Android系统可谓博大精深，加上自己才疏智浅，书中难免存在缺点与漏洞，欢迎大家提出宝贵意见，并不吝赐教。

最后默默期许，有读者赞一声“这书对Android驱动开发还有些用”，本人则不胜感激与欣慰！当然，还要感谢张国强先生及我的家人，正是他们的支持与鼓励，才得以让我坚持并最终完成了这本书。同时，还要感谢网上不知名的“大牛们”给予的无私帮助，正是你们，让我相信中国的嵌入式驱动工程师队伍一定会强大。

本书内容安排

本书的内容被划分为三篇，共20章。

第一篇是Android概述篇，讲述了Android的来龙去脉、Android软件体系结构。它包含了两章内容：

第1章Android的前世今生，讲述了Android的起源、现状与发展。

第2章Android体系结构，讲述了Android的4层软件架构、源代码目录组织及其开发环境的搭建。

第二篇是Linux驱动基础篇，讲述了Linux内核基础知识，以及Linux驱动开发的关键技术。它包含了11章内容：

第3章Linux内核综述，讲述了Linux OS基本概念、进程管理、内存管理与文件系统。

第4章Linux内核编程与内核模块，讲述了Linux内核模块模型、内核模块编程，以及Linux内核源码组织与编译。

第5章Linux文件系统，根据Linux“一切皆文件”的核心要旨，着重讲解了文件系统，特别是与设备驱动紧密相关的设备文件系统。

第6章Linux字符设备驱动，讲述了字符设备驱动程序结构，并实现了一个虚拟化的字符设备，最后讲解了如何通过设备文件名和设备驱动程序来访问设备。

第7章Linux设备驱动中的内存与I/O访问，讲述了Linux设备驱动通过内存映射或分配I/O地址，实现对设备访问的基础原理。

第8章Linux设备驱动中的中断，讲述了Linux驱动的主处理流程与中断处理异步机制，以及中断响应与定时器技术。

第9章Linux设备驱动中的并发，讲述Linux驱动中并发访问存在的原因，以及解决并发竞争的同步访问技术。

第10章Linux设备的阻塞式与非阻塞式访问，讲述了Linux向上层应用提供的阻塞式与非阻塞式两类同步访问设备模式，以及所要提供的相关支撑技术。

第11章Linux设备驱动中的异步访问，讲述了Linux向上层应用提供的非同步（即异步）访问设备模式，以及Linux所要提供的相关支撑技术。

第12章Linux块设备驱动，讲述了Linux块设备的I/O操作与字符设备的不同之处、块设备驱动结构，以及关于块设备驱动的相关支撑技术。

第13章Linux网络设备驱动，讲述了Linux网络设备驱动架构体系、该类设备驱动结构，以及针对该类设备的I/O实现。

第三篇是Android驱动实践篇，讲述了Android HAL等Android驱动开发专有基础知识，以及若干个实践着Linux驱动的Android功能子系统，为Android驱动开发提供具体的帮助与指导。它包含了7章内容：

第14章Android HAL层的设计，讲述了Android HAL层工作原理，并以虚拟驱动为例，列举了针对具体驱动实现HAL层的实例。

第15章Framebuffer子系统，讲述了Android基于Framebuffer实现显示输出的工作机理；从开发实践出发，描述了Android Framebuffer子系统中经常遇到的相关硬件和相关驱动开发；还从系统角度讲解了所开发的Framebuffer驱动如何与Android Framebuffer子系统融为一体，为Android用户提供所需的显示输出服务。

第16章Input子系统，讲述了Android Input子系统的工作机理；从开发实践出发，以扩展键盘驱动为例，描述了Android Input子系统中相应驱动的开发；还从系统角度讲解了键盘等Input类驱动如何与Android Input子系统融为一体，为Android用户提供所需的输入服务。

第17章V4L2子系统，讲述了Android V4L2子系统的工作机理；从开发实践出发，以OV5642 Camera驱动为例，讲述了Android V4L2子系统中相应驱动的开发；还从系统角度讲解了Camera等V4L2驱动如何与V4L2子系统融为一体，为Android用户提供拍照、录像等多媒体服务。

第18章Binder IPC通信子系统，讲述了Android这个专有轻量级进程通信子系统的工作原理；描述了该子系统的Binder驱动底层支撑技术；还从系统的角度讲解了Binder驱动如何与Binder IPC子系统融为一体，为Android中的应用、服务等进程提供进程间通信。

第19章USB子系统，讲述了USB协议的基础知识；描述了Android中USB底层驱动支撑技术；以USB Mass Storage为例，讲解了USB驱动如何与USB子系统融为一体，为Android用户提供USB相关服务。

第20章Bootloader引导子系统，讲述了Bootloader的工作机理；从开发实践出发，给出了若干Bootloader的修改指导。

本书特色

本书立足于Android驱动开发实践，从开发者角度出发，主要有以下几个特点：

• 实战性强。体现在两个方面，一是本书使用了大量的代码和例子，其中有的代码和例子稍做修改就可运用到具体的Android驱动开发中；二是讲解相关技术内容时，处处站在开发者角度，整篇的编排都是根据作者多年开发经验、从实战出发合理组织的。

• 循序渐进。本书内容尽量由浅入深，逐层推进，力求让稍有软件和操作系统基础的入门者也能看懂并能掌握相关的Android开发技术。

• 整体把握。本书始终强调避免将Android驱动与Android的其他组件部分割裂，而是从软件工程系统的观点，确保开发的驱动能成为Android的有机组成，为Android用户提供实际的功能服务。

第一篇　Android的前世今生——Android概述篇

本篇由两章构成：第1章讲述Android的前世今生；第2章讲述Android体系结构。该篇主要是想让读者对Android的历史有所了解，对Android的整体架构有所把握。

第1章　Android的前世今生

Android是一个可用于手机、平板电脑和笔记本等移动设备的、嵌入式的、实时的智能操作系统。Android系统由Google公司在Linux内核和GNU软件基础上开发，上层通过修改JVM，实现了一个安全、易移植的平台。

本书主要讲解Android底层驱动的开发与应用，但也有必要先了解一下Android的前世今生，以便对Android有一个全面、感性的认识。

1.1 Android的起源

Android系统最初由美国一家名字叫 Android的小型创业公司开发，Google公司在 2005年 7月收购了 Android公司。 Android公司的联合创始人 Andy Rubin、 Rich Miner、 Nick Sears和 Chris White也一起到 Google公司工作。 Andy Rubin加入 Google公司后担任 Android项目的负责人，组织开发这个基于 Linux内核、功能灵活、升级方便的移动操作系统。也正是从那个时候起，业界传出了 Google公司打算进入移动手机市场的消息。

1.2　开放手机联盟

开放手机联盟（ Open Handset Alliance， OHA）由 Google公司在2007年 11月 5日正式宣布成立，并且随后在开放手机联盟的旗下公布了全新的 Android操作系统。开放手机联盟是由全世界顶尖的硬件、软件和电信公司组成的联盟，致力于为移动设备提供先进的开放式标准，以便开发可以显著降低移动设备与移动服务开发成本的技术。中国三大电信运营商中国移动、中国电信和中国联通都是开放手机联盟成员，中国移动还是开放手机联盟的创始成员。图 1-1列出了 OHA的部分成员。

图1-1　OHA部分成员

1.3 开源与相关协议

Android操作系统于 2008年 10月 21日在 Apache Software License（ ASL）协议下开放源代码， Google公司在该协议下公布了Android系统的全部源代码。 Google公司选择 Apache Software License 2.0许可证既保证了系统的开放源代码，又鼓励开源软件的商业性使用。 Google公司将 Android置于 ASL许可证之下，可以确保许多商业性公司接受这个平台，并且在它的基础上使用自己的专有技术。图 1-2就是 Apache基金会的标志。

图1-2 Apache基金会的标志

Google公司开放 Android源码，对于 Android开发者与学习者来讲，真是莫大的福音。它为我们学习与研究 Android内在运行机制提供了方便；以及在开发 Android产品过程中，对于产品设计、问题解决等提供了最可依赖的基础。

1.4 系统的升级与发展

系统升级是 Android的一大特色，每次新系统的发布， Android功能和用户体验都有很大提升。而且有趣的是，从 Android 1.5版本开始，每个系统版本都被冠以一个以美国传统食物命名的代号，比如 1.5（ Cupcake）、 1.6（ Donut）、 2.1（ Eclair）、 2.2（ Froyo）、 2.3（ Gingerbread）、 3.0（ Honeycomb）、 4.0（ Ice Cream Sandwich）。细心的人很容易发现，这些食物名称的首字母是以英文字母表为顺序的。更让我们感到亲近的是， Google公司还将所有发布的 Android系统代号做成卡通模型，并将这些卡通模型放在位于加州山景城的 Google公司总部的草坪上，陪伴在 Android小绿人旁边。本书编写完成时 Android最新的版本已是 4.3。

为了方便后面的探讨，我们将基于 Android公司 2.3.7，以及该版本所用的 Linux 2.6.35内核展开。因为 Android 2.3是 Android发展中较为成熟的一个产品。而且读者可以发现，我们只要深刻理解了 Android 2.3，再去分析和理解其他 Android版本并基于这些版本做相应的开发，就没有什么难度。

随着 Android系统手持终端占有率的不断攀升， Android系统吸引了一大批程序开发者不断开发程序来扩展 Android手机的功能。目前 Android电子市场（最新更名为 Android Player）拥有数不胜数的应用程序。另外还有许多设备商、网络 SP公司以及集成商等也纷纷推出自己的 Android App Store，以向用户提供功能强大的各种 Android应用。目前支持 Android系统的 CPU硬件平台最多，市场上 Android手持终端的品类也是最多的。

我们可以清楚地看到， Android已具有良好的生态环境，形成了一个巨大的产业链。通过了解 Android的前世今生，我们应该能感受到，学习与掌握 Android是值得的。希望本书接下来的内容能在读者学习与研究 Android的道路上提供帮助。

第2章　Android体系结构

从本章开始，我们将真正进入关于Android的学习。本章主要介绍Android的体系结构。希望通过本章的学习，读者能对Android有个整体的了解。本章在本书中不是重点，但仍请读者偶尔回头翻一翻该章的内容，因为这样让我们在畅游繁杂的Android系统时不会迷失了方向。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

2.1　四层空间基本结构

Android是一个开放的软件系统，从下至上包括 4个层次，如图 2-1所示。

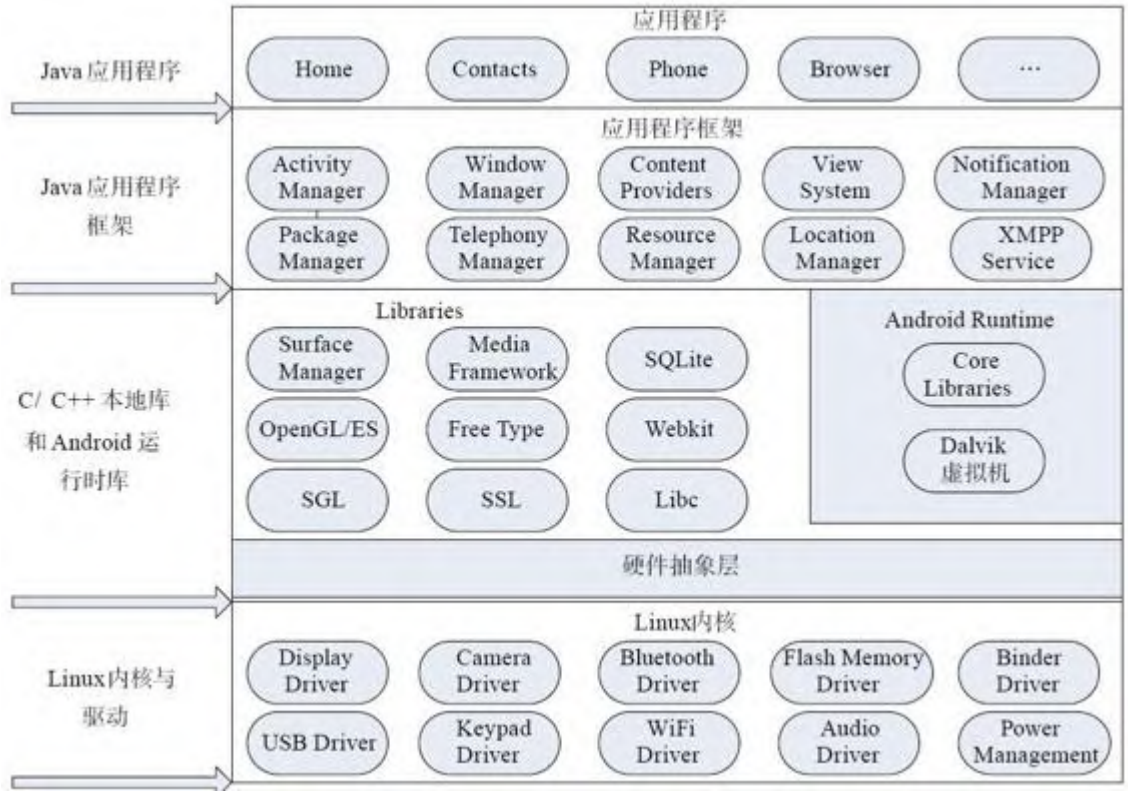


图2-1　Android四层空间架构

其中第一层是 Linux内核层，包括 Linux操作系统及驱动。不同的 Android版本所用的 Linux内核版本不一样；甚至在不同平台，即使所用的 Android版本一样，其所用的 Linux版本也会有差异。例如，Android 1.6可能用的是 Linux 2.6.29；而 Android 2.3.7则可能用的是 Linux 2.6.35；最新的 Android 4.x则纷纷采用 Linux 3.x。虽然， Linux 2.6.1之后的任何一个小版本都会有很大变化，但我们认为，就目前 Android所用的 Linux版本来讲，这些差别并不会有我们想象中的那么大，所以在第二篇中不会太追究 Linux版本之间的差异，而是关注最根本的东西，让读者能有所收获，并打下一个扎实的基础。 Android基于 Linux内核，还增加了 Binder驱动等模块，为系

统运行提供了 IPC进程通信等方面的支撑。我们将在第三篇对关于Android驱动的特性部分展开更详实的描述。

第二层是核心的扩展类库，如 SQLite、 WebKit、 OpenGL等，它们可以通过 Java本地调用（ Java Native Interface， JNI）的接口函数实现与上层之间的通信。该层由 Android的 Java虚拟机 Dalvik和基础的 Java库为 Java运行环境提供了 Java编程语言核心库的大多数功能。

第三层是包含所有开发所用的 SDK类库和某些未公开接口类库的框架层，是整个 Android平台核心机制的体现。

第四层是应用层。系统自带应用和第三方开发的应用都位于这个层次上，但两者不完全相同：其中系统应用会使用一些隐藏的类，也就是说，这些类没有包含在 SDK中；而第三方开发的应用，是基于 SDK基础上开发的。一般 Android开发是在 SDK基础上用 Java编写应用程序，但本机开发程序包 NDK提供了应用层穿越 Java框架层直接与底层包含了 JNI接口的 C/C++库直接通信的方法。在研发工作过程中，我发现有人常将 NDK与 JNI混为一谈。从这里，我们应认识到它们还是有所区别的： Android的应用开发者用 JNI与 C/C++库通信的话，他应该通过 NDK；而对于本书的主要读者—— Android的驱动开发者来讲，则主要考虑如何为底层驱动实现 JNI接口，以便上层可以方便地调用由该驱动程序所驱动的相应设备的功能。

从 Linux操作系统的角度来看，第一、二层次之间是内核空间与用户空间的分界线，第一层运行于内核空间，第二、三、四层运行于用户空间。第二、三层之间是本地代码层和 Java代码层的接口，第三、四层之间是系统 API接口。

另外，用心的读者也许会从图 2-1中发现，在第一、二层之间，即第二层的最底层，有一个硬件抽象层（ Hardware Abstract Layer，HAL）。其实，该层并不一定要存在，它是为了保护一些硬件提供商的知识产权而提出的。根据 Android的 Apache License，硬件厂商可以只提供二进制代码，但这些二进制代码应该符合 HAL规范，以便上层调用这些硬件所提供的特殊功能。具体内容我们将在第三篇的第 14章进行详细的探讨。

基于上面这个分层结构，我们所从事的 Android开发主要分成两类：一类是 Android系统底层开发；另一类就是 Android应用开发，如图2-2所示。在本书中，我们将主要关注 Android系统底层开发。但为了让底层开发能为 Android应用开发所用，我认为底层开发者至少应对上述 Android四层架构中的第二层与第三层有所了解。在必要的情况下，比如一个新的设备模块，就会要求我们提供相应的 C/C++库、JNI，甚至应用 Demo等，以便上层应用基于我们的设备开发出相应的功能提供给最终用户。

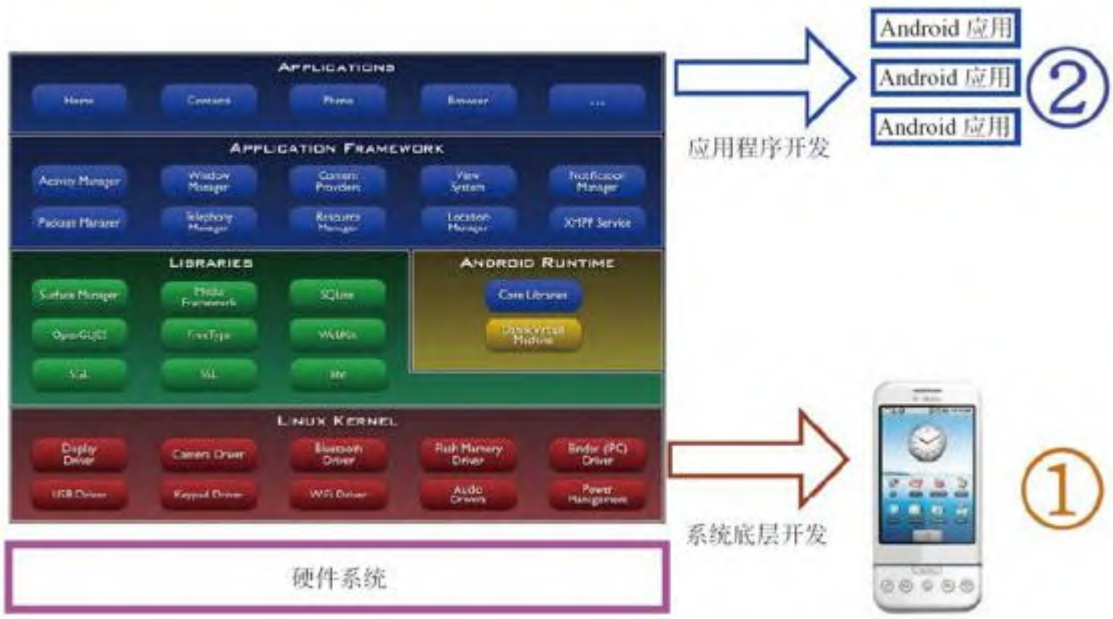


图2-2　Android开发分类

2.1.1　Android系统底层开发

我们将除第四层之外的所有层的开发都称为底层开发。其中第三层是Java框架层，我们一般不做修改；在第二层，我们会为调用底层硬件功能编写 JNI、 HAL以及相关的动态共享库。如图 2-3所示：图中Java应用程序、 Java框架、核心库等部分是不会修改的； JNI、Android各种底层库、硬件抽象层等部分将会有一定的修改；而处在内核空间的各部分，尤其是 Android系统相关设备驱动部分将会有大量的修改甚至新的创建在里面。

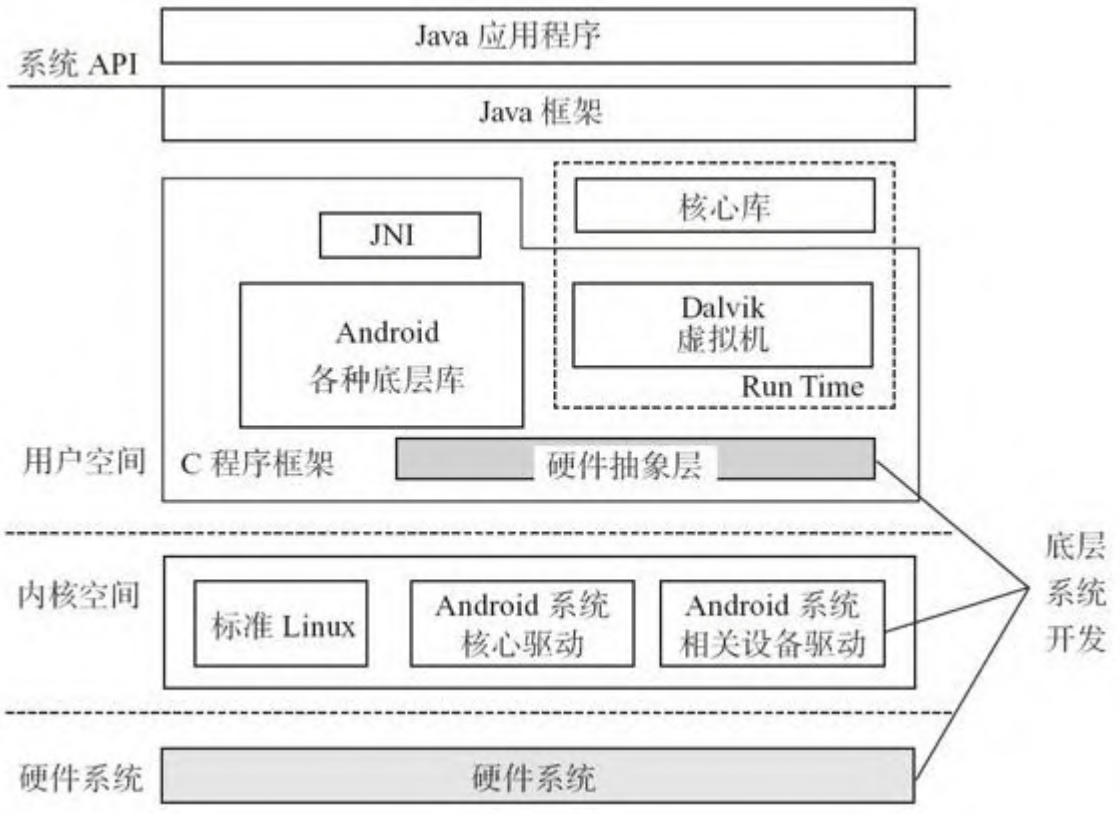


图2-3　Android底层开发

2.1.2　应用程序开发

应用程序不是本书关心的部分，因此这里了解一下就可以。如图 2-4 所示。

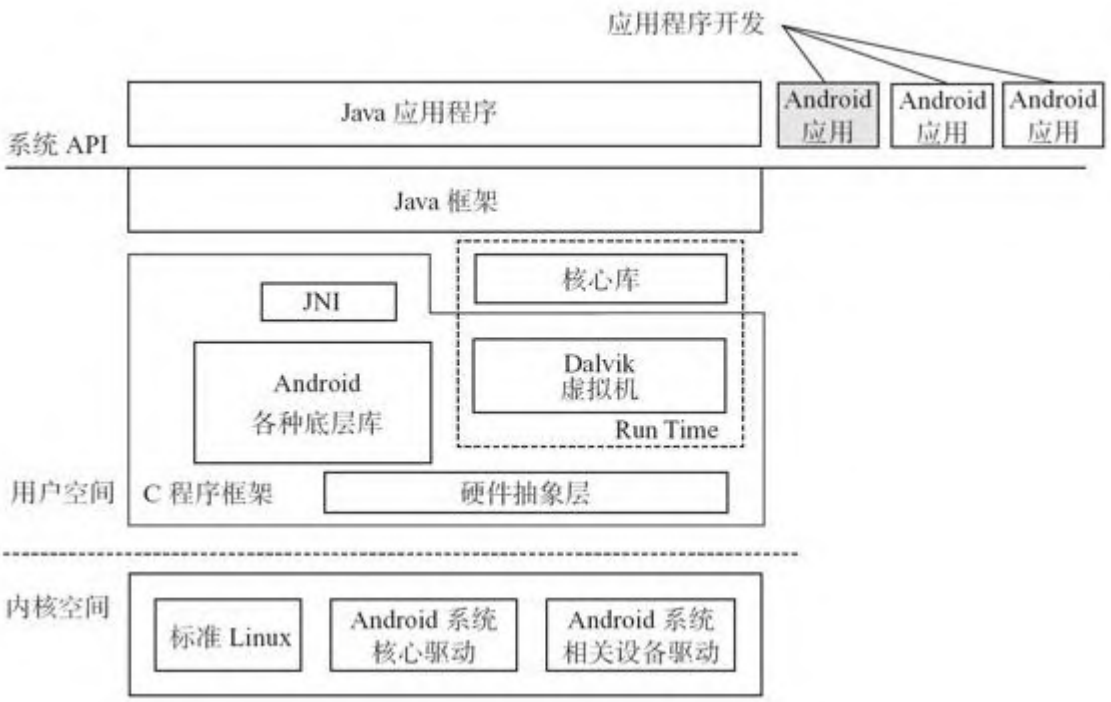


图2-4　Android应用开发

对这部分感兴趣的读者，可以参考 http://developer.android.com/index.html，里面有针对应用开发各部分的讲解，并有翔实的例子。

2.2 Android代码目录结构

我们还应了解 Android的代码组织方式，这样将对阅读代码和分析问题有很大的帮助。

先介绍 Android源码阅读与开发的前提条件：因为对于前面所述四层结构中，第一层由 C语言实现，第二层由 C和 C++实现，第三、四层主要由 Java代码实现，所以，要学习 Android开发，应具有 C、C++、 Java三种语言的基础。

Android代码包括三个部分：①核心工程（ Core Project）是建立 Android系统的基础，在根目录的各个文件夹中；②扩展工程（ External Project）即使用其他开源项目扩展功能；③包（ Package）提供 Android应用程序和服务，其中既包含要在 Android设备上运行的代码，还包括主机编译工具、仿真环境等。

第一级别的目录和文件如下：

1） Makefile：全局的 Makefile。

2） bionic：这里面是一些基础的库的源代码。

3） bootloader：引导加载器，不同的平台，该名字会稍有不同，但其意义差不多；事实上，有很多平台都有专有 boot目录，以满足这些平台特殊的引导要求。

4） build：目录的内容不是目标所用的代码，而是编译和配置所需要的脚本和工具。

5） dalvik： Java虚拟机。

6） development：程序开发所需要的模板和工具。

7） device：与实际目标机器所用平台直接相关的库或源码。

8） external：目标机器使用的一些库，其实现了开源的扩展功能。

9） frameworks：应用程序的框架层。

10） hardware：与硬件相关的库，前面讲的 HAL就是在这个目录下实现的。

11） kernel： Linux的源代码。

12） packages： Android的各种应用程序。

13） prebuilt： Android在各种平台下编译的预置脚本。

14） recovery：与目标的恢复功能相关。

15） system： Android底层的一些库。

16） vendor：有的 CPU厂商用它来放置与 CPU相关的 CSP或 BSP相关代码或库。

注意，上面有的目录可能在相应的 SDK中不存在，或者在 SDK中有的目录该列表中没有，这要看具体开发的机器与平台，但一般 SDK主体就是上面 16个目录。

另外，这里再解释几个我们经常碰到的名词： SDK全称是 Software Develop Kit，是指 Android开发完整的软件包，是对上述代码的总称； CSP全称是 CPU Support Package，是针对某操作系统而适配的、与 CPU紧密相关的代码与库； BSP全称是 Board Support Package，是针对某操作系统而适配的、与具体开发的 PCB板紧密相关的代码与库； PCB全称是 Print Circuit Board，一般是指印刷电路板，它是开发将要依赖的硬件基础。

编译完成后，将在根目录中生成一个 out文件夹，生成的所有 Android代码结构内容均放置在这个文件夹中。 out文件夹包含如下目录和文件：

1） CaseCheck.txt。

2） casecheck.txt。

3） host。

4） common。

5） linux-x86。

6） target。

7） common。

8） product。

其中，两个主要的目录为 host和 target，前者表示在主机（ x86）生成的工具，后者表示目标机（如 ARMv5）运行的内容。

2.3 Android开发环境搭建

我们要进行 Android开发，首先要有一台 PC。 PC的最低配置如下：

1） 32位 4核 CPU。

2） 2GB内存。

对于 Android 2.3.x之后的版本， Google公司逐渐要求 64位的 CPU，而且要求内存在 8GB以上。否则用虚拟 64位机的方式，仅编译的时间就让人无法忍受。但是这么高的机器配置，可能让 Android开发爱好者望而却步。

不过，我们找到了一种用上述低配 PC就可以编译 Android 2.3.x的方法。网上有很多相关的描述，这里不对该方法做更多的讲述了。这也是本书选定 Android 2.3.7为蓝本的主要原因之一。另外基于低配的 PC，用 Eclipse调试 Android 4.x以上的应用，也是相当慢的。因此，我们建议 Android开发初学者，不妨先从 Android 2.3.x入手。笔者认为，学习一门技术，特别是像 Android之类的系统级软件，掌握其思想是最重要的。

有了 PC硬件之后，还要为该 PC准备好 Android开发软件环境。该环境的搭建分成两大步。

1） Ubuntu Linux的安装。这一步又可以分为两小步：

① VMware虚拟机的安装。可能对于大多数的 Android开发初学者来讲， PC上应该预先装载了 Windows系统；为了安装 Ubuntu Linux，我们应该在 Windows系统上安装虚拟机。而虚拟机软件，我们推荐 VMware。对于专业的 Android开发者来讲，建议反过来，系统默认安装 Ubuntu，而 Windows系统安装在 Ubuntu的虚拟机上。

② Ubuntu的安装。建议使用 Ubuntu 10.04及以上版本。推荐使用 10.04，因为 Google以及各 CPU原厂都是基于该版本来开发验证的。另外，分配 Ubuntu的硬件空间建议尽量大于 90GB，因为 Android的开发包太占空间了。如果 PC的内存足够大，也尽可能为 Ubuntu分配更多的内存空间，这样会大大缩短编译时间。

2） Android编译环境搭建。这一步则可以分成以下几步。

① Repo工具安装。执行以下两步命令：

```
$ curl https://dl-ssl.google.com/dl/googlesource/gitrepo/repo
>~/bin/repo
$ chmod a+x ~/bin/repo
```

该工具将被用来从 Google公司指定的网站提取 SDK源码。如果第一条指令不成功，不妨将 https修改为 http试试，或者安装 ssl相关的包，以让 Ubuntu支持 https。

②安装 SUN JDK1.6。 Ubuntu中已不默认支持 JDK安装。你不妨直接从 Oracle网站直接下载一个，并执行以下命令直接解包即可：

```
$ ./jdk-6u37-linux-x86.bin
```

注：上面的数字 37是该 JDK 6.0的版本号，用户下载的版本不同，该数字可能不同。

如果你安装的是 64位的 Ubuntu，则运行的应该是 jdk-6u37-linux-x64.bin。

接着，要配置 Java的环境，以便 Ubuntu中的其他工具或软件可以方便地用到 Java。因此，我们修改了文件 etc/profile，在该文件的最后 3行加上以下代码：

```
export JAVA_HOME=/root/tools/JDK/jdk1.6.0_37/
export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH
```

最后，执行下面的命令，来验证 Java的版本以及是否安装成功：

```
$ java -version
```

③安装必要的其他工具。执行以下的命令即可：

```
$ sudo apt-get install git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs x11proto-core-dev libx11-dev lib32readline-gplv2-dev lib32z-dev libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-utils xsltproc
```

④升级 Git工具。对于 Ubuntu 10.04， Git工具默认的版本有些旧。到 Android 2.3.x版本之后， Git版本要求 1.7.1之后的。从网上下载一个最新的 Git包，解包并执行以下的命令即可：

```
$ tar -zxf git-1.7.1.tgz
$ cd git-1.7.1
$ ./configure
$ make
$ sudo make install
```

升级完成后，可以用下面的命令来验证升级的成功：

```
$ git --version
```

当开发环境搭建好后，就应针对目标机做工作了：那就是 SDK源码的获得，以及编译。编译好的 Image将被烧写进目标机。

1） SDK源码的获得。执行以下命令，这要花费不少时间。

```
$ mkdir [android_dir]
$ cd [android_dir]
$ repo init -u
https://android.googlesource.com/platform/manifest
$ repo sync
```

2） SDK编译。笔者认为， Android的编译应该分成 3部分（平台不一样，可能会有些许差别）：

① uboot的编译。该编译所产生的 Image将用来引导 PCB板，装载与下载后面的 Kernel与 Android的相关 Image。

② kernel的编译。就是对 Linux内核的编译，它与一般的 Linux编译没有太大的区别，其所生成的 Image，将用来承载 Android所需要的 Linux OS。常用到的编译步骤有：

```
$ make menuconfig      #对Linux内核各组成模块进行配置
$ make                 #编译生成Linux内核映像
```

---

③ Android的编译。其用来编译生成 Android的相关 Image。一般主要有 3个： system.img、 ramdisk.img、 userdata.img。编译的方法就是在 Android的主目录下，执行以下的命令：

```
$make
```

生成完 Image后，我们就应找一块开发板，来验证我们的编译成果。我们强烈建议读者想办法找到相应的开发板来检验开发。如果实在找不到，不妨把生成的 Image文件复制替换 Android AVD的相应文件，并用 Eclipse来调试自己的修改。请记住，我们讲的是 Android底层开发技术，这是个实践性很强的技术，有机会就要多动动手。

第二篇　勿于浮砂筑高台——Linux驱动基础篇

下面将讲述Android最底层的部分：Linux OS。Linux OS其实是一个很大的课题。这里我首先要提醒各位读者，学习Linux OS的目的是让用户可以方便地使用PCB板上的CPU，以及GPS等功能模块。因此我们在着手开发驱动程序之前，应先想到相应的功能设备，想想我们开发该驱动的目的何在。我们将在本篇让读者对Linux OS有一个基础而清晰的认识。

第3章　Linux内核综述

如前面所讲，Android的OS（Operation System，操作系统）是Linux。因此，如果对Linux内核没有一定的理解，就去进行Android开发，甚至Android底层开发，哪怕是在“前辈”的基础上对某个模块进行修修改改得到了想要的结果，也终究走不太远。老祖宗教导我们“勿于浮砂筑高台”，诚不欺也。

对于具备一些Linux基础的读者，不妨跳过本篇中的某些章节。但还请大家有空的时候，多多阅读这部分的内容。完全没有基础的读者，也不用担心，安心地阅读下去，相信一定会有所斩获的，因为我们尽力用浅显的方式来捅破Linux的一层层窗户纸。

3.1 OS基本概念

现在的 PC和常用的手机等，都有一组基本的程序—— OS（操作系统）。在这组程序中最重要的叫内核。内核将在系统引导时被装载进RAM，其中包含了许多关键的例程，以操作系统；当然除了内核之外，OS中还有其他的程序以方便用户交互，方便用户使用计算机。可以说，内核是 OS最为关键的部分，人们常将 OS与内核等同。 Android设备中所用的 OS就是 Linux 2.6之后的版本。在本书中，除非特别说明， OS就是指 Linux内核。并且，本篇将基于 Android 2.3所用的Linux 2.6.35的内核来讲解。

OS必须完成两个主要目标：

- 与硬件交互。
- 为在计算机系统上运行的应用程序提供可执行的环境。

一些 OS允许用户程序直接与硬件组件交互（如 MS-DOS）。相反，Linux OS会屏蔽掉所有与物理硬件相关的东西。当应用程序要用到硬件资源时，它就必须向 OS发出请求，而 OS则会根据情况许可上层应用使用该硬件资源。

为了加强这种机制，现代的 OS将依赖于特定硬件的特性（ feature）来禁止用户程序直接与底层的硬件组件交互，或直接地随意访问内存的地址。继而，该硬件引入了两个不同的 CPU执行模式：非特权模式给上层用户程序用；特权模式则给内核用。相应地， Linux中将之称为用户模式和内核模式。

下面我们先了解关于 Linux的几个概念。

3.1.1 多用户系统

Linux OS是源于 UNIX系统的。 UNIX设计时，考虑到计算机是很昂贵与稀缺的资源，因此一台计算机就要满足多个用户同时使用。 Linux OS沿用了这一设计思想，就通过分时共享等策略，让多个用户可以同时使用一台计算机，却让各用户感受不到自己是在与其他用户共用这

台机器。该分享策略，使得机器即使只有一个用户，也可以同时运行多个任务，响应多个进程。

Linux作为一个多用户系统，它必须具有以下几个特性：

- 认证机制，以验证用户ID。
- 保护机制1，以对抗bug，不让这些坏程序阻塞了其他用户程序。
- 保护机制2，以对抗恶意的程序，以防它去监听其他用户的活动。
- 审计机制，以限制分派给每个用户的资源。

为了确保机器的安全， Linux内核要求使用由 CPU硬件提供的特权模式；而用户程序只能运行在非特权模式下；用户程序要使用底层的软、硬件资源，必须向 Linux内核发出请求，在 Linux内核许可后，并切入特权模式以执行。提醒各位读者一点，要实现这两种模式，常常需要 CPU在硬件上给予支持。例如，在现在最流行的 ARM处理器中， CPSR（当前程序状态寄存器）就用了 5个位来识别处理器的 7种模式，其中两种模式就与 Linux中的用户模式和内核模式相对应，其他 5种模式则用来标识不同的中断或出错场景。

3.1.2 用户和组

Linux OS作为多用户系统，每个用户在机器上都有一个私有空间；特别是，他会拥有配额的磁盘空间（对于 Android手持终端来讲，更可能是 Flash空间或 MMC存储空间），以存储文件、接收私有的 mail消息等。 OS必须确保这部分私有空间只对它的拥有者是可见的。另外，它应该确保没有用户可以使用系统应用程序来侵犯其他用户的私有空间。

所有用户都是通过唯一的用户 ID来识别的，简称 UID。当用户想用机器，会被要求输入用户名和密码，这样用户的私密权就得到了保证。

如果要选择与其他用户共享材料，共享的每个用户就应是一个或多个组的成员，这个组由组 ID（ GID）来识别。每个文件都可与一个确切的组相关联。例如，作为文件拥有者的用户拥有对该文件的读写权

限，而组中其他用户则只拥有读权限，系统中非组中的用户则没有任何访问权。

Linux OS中，有一个特殊的用户： root。系统管理员应以 root的身份登入机器，以管理其他用户账号，完成维护任务（如系统的备份和程序的升级等）。 root用户几乎可以做任何事情，因为 OS并没有对它采取一般的防护机制。另外， root用户可以访问系统上的每一个文件，可以管理每一个正运行着的用户程序。

如果需要，在 Android开发中，可以通过超级终端或 adb shell，可以输入以下命令来拥有 root的权限：

```
$ su -
#
```

3.1.3 进程

前面我们提到了进程。事实上，对于支持多用户的 OS都有一个基本的抽象：进程。进程是一个执行的实例（ an instance of a program in execution），是程序执行的上下文（ execution context）。在初始的 OS中，一个进程是在一个地址空间中，执行单个顺序的指令集。现代的 OS中，则允许进程中有多个执行流，也就是在同样的地址空间里有多个同步运行着的顺序执行指令集。作为 Linux OS，它就允许进程中有多个执行流可以并发执行。这些执行流在 Linux OS中被称为线程或轻量级进程。

对于单处理器，进程、线程或中断处理执行流的并发，是分时的并发，是逻辑上的并发。对于多处理器系统，这些并发执行是真真切切同时在运行的。多用户系统，必须保证有一个这样的并发执行环境：允许几个进程同时执行，它们将对系统的资源共享，且通过竞争来获得使用。这些允许多个活动进程的 OS，也被称作多程序或多进程系统。 Linux OS就是支持多用户、多进程、多处理器的一款操作系统。

注 1：程序与进程是有区别的。程序是静止的，由代码、库或资源组成的集合体；而进程则是程序在计算机系统里运行，是程序的动态体现。

注 2：有些多进程系统并不是多用户系统，比如 Windows 98。

OS中有一个组件叫调度器（ scheduler），它将负责选择可以被执行的进程。这个选择的过程，我们称之为进程调度。支持多用户的 OS要跟踪某一进程已拥有 CPU多长时间了，从而周期性地激活 scheduler，从而对系统的进程重新进行一次调度，以便其他用户能够得到执行的机会。

Android所用 Linux 2.6是一个多进程的系统，同时实现了可抢占式的进程调度，也就是高优先级进程可以抢占低优先级进程的 CPU执行权，以便高优先级的任务能得到更快的响应，从而满足 Android系统的实时性要求。 LinuxOS即使没有用户登录或没有应用程序运行，其实也会有几个系统进程在运行，以监控机器的外围设备。特别是有几个进程会监听系统的终端以等待用户的登录。当有用户输入了用户名，该监听进程就会去核查用户的密码是否正确。如果用户的 ID被认可了，该监听进程就会创建另一个进程。这个监听进程就是以命令行形式与 Linux交互的 Shell进程。 Android系统启动后，肯定会运行 Launcher应用，它其实是一个图形化 Shell，从这里用户可以点击其他应用和 Icon，从而激活其他进程执行相应的程序。

3.1.4　Linux单核架构

OS的内核架构主要分为两种：一种是单内核的；另一种是微内核的。Linux内核是单核的：每个内核层被集成为一个整体的内核程序，并运行在当前进程的内核模式中。相反，微内核 OS则只有上述内核的最核心功能，一般包括：一些同步用的原语、调度器、与进程间通信的机制；而几个系统进程会运行在上述微内核之上，来完成 OS其他系统层的功能，如内存分配、设备驱动，以及系统调用处理例程（ Handlers）。像微软的 Windows CE OS就是微内核的。

尽管学术界更倾向于微内核，但微内核 OS速度上较单核的架构要慢，因为在 OS不同层之间的显式消息传递就是要花费时间的。当然，从理论上讲，微内核具有许多单内核所不具有的优点。微内核强制系统程序员采用模块化的方法，因为每个 OS层是一个相对独立的程序，它要通过已定义好的、简洁的软件接口与其他层交互。另外，微内核的 OS移植起来将更简单，因为所有与硬件相关的组件基本都被封装在微内核代码中了。最后，微内核 OS相对于单内核 OS，能更好地使用 RAM，因为那些功能不为所需的系统进程可以被交换出去甚至毁灭。

为了得到上述微内核理论上的好处，却又没有性能上的损失， Linux内核基于面向对象的思想，引入了模块的概念。一个模块包含一个object文件，该 object代码由一组功能组成：归属于某一文件系统、底层设备功能驱动，以及其他属于内核上层的特性（ features）。这个模块又不同于微内核 OS的外部层，它不是运行在专门的进程中，而是运行在当前进程的内核模式中。本书所要讲述的重点—— Android底层驱动，就是以模块的方式实现的。

总之， Linux OS使用模块机制主要带来了以下好处：

- 模块化的方法。
- 保持平台的独立性。
- 节俭的主内存使用。
- 没有任何性能上的损失。

为使大家有一个感性认识，下面实现一个 hello模块，参见代码清单3-1。

代码清单 3-1　hello模块

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT"Hello, Android Driver");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT"Goodbye, Android Driver");
}
module_init(hello_init);
module_exit(hello_exit);
```

上面的代码很简单，其主要功能就是当调用 insmod命令，向系统内核插入该模块时，输出 "Hello， Android Driver"，而调用 rmmod命令从系统卸载该模块时，输出 "Goodbye， Android Driver"。 Linux驱动，包括 Android驱动，就是该类模块的一种。在第 4章，我们将对Linux模块机制展开更详细的讲解。有兴趣的读者可以就这个例子模块，在 PC主机的 Linux系统、 Android设备或模拟器上测试验证。

3.2 Linux内核综述

内核必须实现一组服务和相应的接口，应用程序则可以用这些接口，而不是直接与硬件打交道。下面就针对这些服务，进行一次总体的描述，帮助读者理清这些服务的出发点并了解相应的概念。

3.2.1 进程 /内核模型综述

Linux内核主要由以下 5个子系统组成：进程调度、内存管理、虚拟文件系统、进程间通信以及设备驱动。如图 3-1所示。

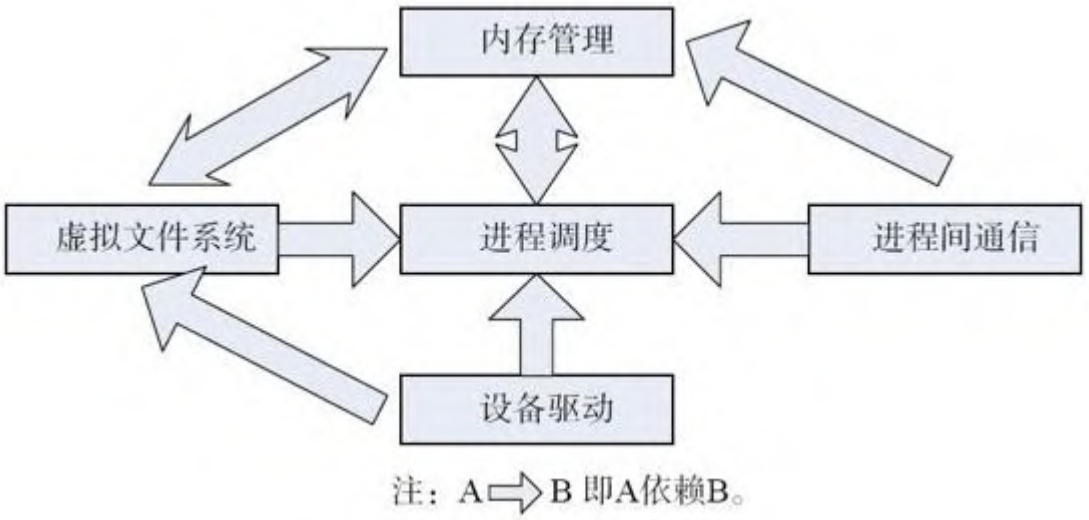


图3-1 Linux内核组成

在这个组成中，其中最核心的就是进程管理，也是图 3-1中的进程调度与进程间通信两个部分。本节将讲述 Linux进程管理的关键特性。

我们首先应了解，正是基于该组成模型使得编写的 Android程序，不管是上层的还是属于中间框架层的代码，甚至是最底层的驱动模块，都可以以进程的形式在系统上运行。前面我们讲了， CPU可以运行在用户模式与内核模式。

在 Linux OS进程 /内核模型中，每个进程就是执行在机器上的唯一的镜像，它们对系统服务的访问具有排他性。当进程需要访问系统服务时，它会发出系统调用（就是对内核的请求），硬件则会将权利模式

由用户模式改为内核模式。此时，该进程就开始执行某内核的过程，这个过程是有严格限制的。通过这种方式， OS（确切地讲应该是内核代码）运行在该进程的上下文（ context）中，却又保证其请求是安全的。该内核过程在合适时机会通过硬件强制返回用户模式，此时该进程将继续执行程序中紧接着系统调用的下一条指令。

当一个程序执行在用户模式下，它就不能直接地访问内核数据结构或内核的程序。当程序处于内核模式时，这些限制就不再存在了。现在CPU一般提供了特定的指令在用户模式与内核模式之间切换。通常，在用户模式下执行的程序向内核发出服务请求时，就会切换到内核模式。当内核已响应程序的请求时，就会设置程序退回用户模式。

进程是动态的实体，其在系统中的生命是有限的。任务的创建、销毁和同步在内核中体现为一组线程。 Linux内核是以线程为调度的单位。

注 1：任务就是进程的集合，其中可以包含一个或多个进程。当用户想让机器做某件事时，就会启动一任务，而该任务可以由一个或多个进程来完成。

注 2：同步与异步。读者对同步与异步要有深入的理解，因为在后面这两个词出现的频率很高。异步，就是两件事的发生是没有任何关联的，一件事的发生不会因另一事件的发生而有任何变化；同步，则是两件事之间有先后顺序之分，一个事件的发生一定是在某个事件之后。例如，当一个程序访问某一硬件资源时，另一个程序也要访问该硬件资源，后一个程序就要等待前一个程序，我们就说这两个程序同步访问该硬件资源。千万不要把同步理解为两件事同时发生，初学者常有这样的误解。

内核不是进程，而是进程的管理者。

除了用户进程， Linux系统包括一些特权线程，这些内核线程有以下特点：

- 它们运行在内核模式，用的是内核地址空间。
- 它们不与用户直接交互。

• 它们通常在系统启动时被创建，并一直存活到系统关闭。

在单处理器系统中，某一时间只有一个进程在运行，该进程可能运行在用户模式，也可能运行在内核模式。如果它运行在内核模式，处理器就正在执行内核例程。图 3-2显示了用户模式与内核模式间的迁移。

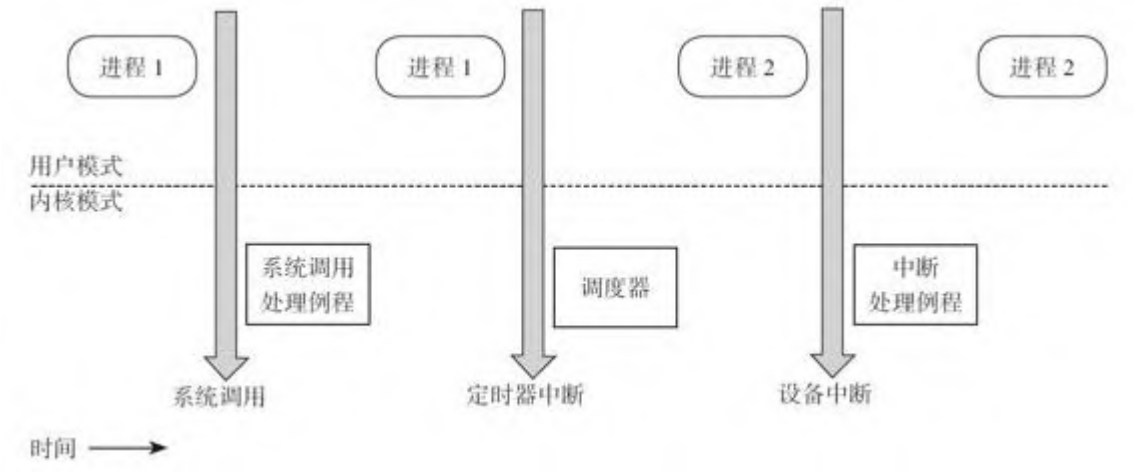


图3-2　进程用户模式/内核模式间的迁移

Linux内核除了处理系统调用，还会做很多事情。事实上，内核例程会在以下情况被激活：

• 进程调用系统调用。

• CPU正执行进程发出异常的信号，表示诸如无效指令等非常规条件发生了。此时内核为了保护引起该异常的进程和计算机，会处理该异常。

• 外围设备发出一个中断信号到CPU，告诉CPU某一事件（如请求注意、状态改变或I/O操作的完成）。每个中断信号都是由一个叫做中断处理句柄的内核程序来处理的。中断与CPU的执行是异步的，所以中断的发生是不可预期的。

• 内核线程被调度执行。

注：异常与中断很相似，它们都会导致 CPU停下当前的进程进入专门的处理例程。但异常是同步发生的，它一定发生在某一错误之后；而

中断则是异步发生的，它的发生是随机的；因此虽然在处理流程上很相似，但 OS还是给予了它们不同的名称。

1.进程实现

为了让内核可以管理进程，每个进程都由一个进程描述符（ descriptor）来代表，在该描述符中记录了该进程的当前状态。

当内核停止了一个进程的执行，它会在该进程描述符中记录几个处理器寄存器的当前内容。这类寄存器包括：

• PC与SP寄存器。

• 通用目的寄存器。

• 浮点寄存器。

• 处理器控制寄存器（处理器状态Word），其中包含了CPU状态的信息。

• 内存管理寄存器，用来跟踪该进程所访问的RAM。

当内核决定重新执行一个进程时，它会用到合适的进程描述符的相关域来装载 CPU的寄存器。例如 PC的信息就会让进程从被暂时中止的位置开始执行。

当进程没有在 CPU上执行，它就应在等待某些事件。 Linux内核区分出许多等待状态，这些状态通常通过进程描述符的队列来实现；每个队列（有可能是空队列）对应着一组进程，这组进程都在等待着特定的事件。也就是讲， Linux会为每个事件创建队列数据结构，该数据结构类型其实是一个链表，该链表上的元素其实就是所有等待该事件进程的描述符。

2.可重入的内核

Linux内核都是可重入的。这就意味着，几个进程可能同时在内核模式下执行。当然，在单处理器系统，在某一时间只会有一个进程运行，但许多会阻塞在内核模式；这些进程会分时共享 CPU、 I/O设备等系统资源；这些进程给用户的感觉就像是在同时运行。

要提供可重入性代码方法是：编写的函数都只会影响到局部变量，而不能改变全局的数据结构。这样的函数都被称作可重入函数。但是一个可重入内核不仅有这些可重入函数，事实上，如上所讲，由于在访问 I/O等共享资源，内核还要有不可重入函数。此时， Linux内核就要使用锁机制，来保证在某一时间内只有一个进程可执行该不可重入代码，也就是分时共享的办法，来实现可重入的内核。

如果某硬件中断发生， Linux内核作为可重入内核，在任何情况下就可挂起该正运行着的进程。这个能力是非常重要的，因为它提高了发出中断的设备控制器的吞吐量。一旦一设备发出了中断信号，它会等待 CPU响应它。如果该内核可以快速地应答，该设备控制器就可以在CPU处理中断时，马上去处理其他事件。在中断处理例程中，不应该有不可重入代码。因为不可重入代码会导致处理流程等待，进而导致中断不能得到及时的处理。更糟糕的是，我们后面会讲到，如果允许中断处理中使用不可重入代码，很容易导致系统死锁而代码得不到进一步的执行。中断处理例程除了可以被更高优先级的中断暂时中止，它应该能被顺序地执行完。

现在让我们仔细看看内核的可重入，以及其对内核组织所带来的深远影响。内核控制路径是指内核处理系统调用、异常和中断的一系列指令集。

在最简单的情况中， CPU顺序地执行内核控制路径，从头执行到尾。但当有以下情况发生时，则 CPU就要交替地执行内核控制路径：

• 在用户模式下运行的某进程调用了系统调用，而当前的内核控制路径检验到该请求不能被马上响应，则内核会调用调度器，并选择新的进程来运行。这样，原来运行的进程被挂起，而一个新进程切换进来拥有CPU的执行。

• CPU检查到有异常发生。例如访问的页面不在RAM中。此时，异常处理例程这个内核控制路径会被执行，但是它与出现异常的内核控制路径是在同一进程的执行上下文中。

• 发生硬件中断。与异常一样，切换的两个内核控制路径也是在同一进程的执行上下文中。

• 更高优先级的进程到来。

图 3-3显示了内核控制路径的交替，此时 CPU会有 3种状态：

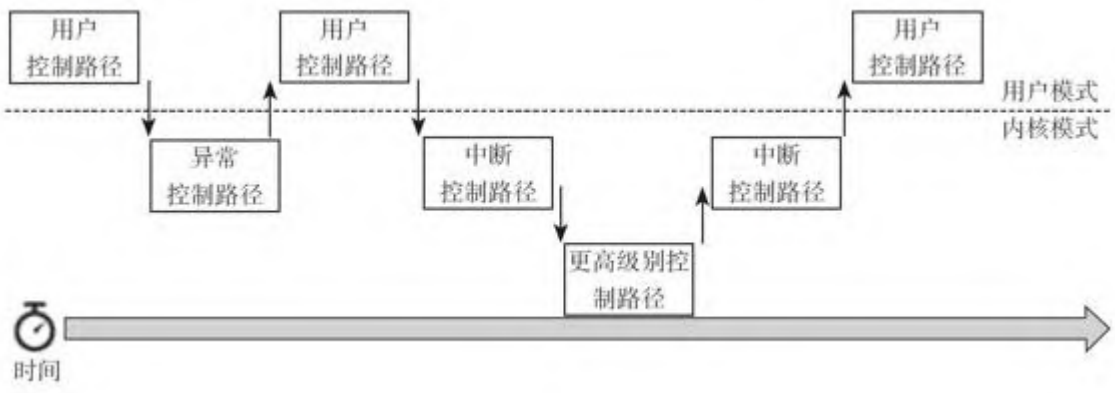


图3-3　CPU的三态切换

- 用户态。
- 异常态（此时有异常产生，或有系统调用句柄在执行）。
- 中断态。

注：上面的描述，还隐含了一个信息：异常处理例程、中断处理例程，以及这些例程的嵌套，当它们中止其他内核控制例程时，是不会变更进程执行上下文的。换句话讲，只有当进程处于用户态时，才会保存被中止进程的状态。

3.进程地址空间

每个进程都运行在其私有的地址空间中。运行在用户模式下的进程会用到私有的栈、数据区和代码区。当运行在内核模式下，该进程寻址的则是内核数据区和代码区，并会用到另一个私有的栈。

因为内核是可重入的，几个属于不同进程的内核控制路径会被轮流执行。在这种情况下，每个内核控制路径都会引用各进程的私有内核栈。

这样对每个进程而言，它们对自己的私有地址空间是有访问权的，并可以将私有空间中特定的部分地址空间公开出来为多个进程所共享。在某些情况下，这共享是由进程显性要求的；另一些情况则是由内核自动使能的，以便节约内存的使用。

如果一个程序为多个用户同时启动，该程序仅会被装载进内存一次。其指令代码为所有需要的用户所共享。它的数据则当然不能共享，因为每个用户都会有其独立的数据。这种指令代码地址空间的共享就是由内核自动完成的，以节约内存。而用户如果想将其数据与其他进程共享，就要在进程中通过显性的指令告诉系统方可。

进程也可以共享它们的部分地址空间，以用于进程间通信。此技术就是共享内存。 Linux支持 mmap（）这个系统调用，以便让某文件的部分或存储在块设备的信息，映射到进程的某个地址空间，这个内存映射将这些内容放置在为所有共享之的进程的地址空间中。其实，内存映射就是将用户地址空间与内核地址空间中的一块关联起来；各进程通过该被关联的内核地址空间，实现相互之间的信息交互。

4.同步与临界区

实现可重入内核的一个关键技术就是同步技术。如果某内核控制路径，因对一内核数据结构访问而被阻塞，则是不允许其他内核控制路径对同样的数据结构进行操作。否则，两个控制路径会破坏该存储的信息，导致内核控制路径执行不确定。

当 CPU计算是两个以上的进程，因访问共享资源而要发生调度，我们就称这里有竞争条件发生了。

在程序中，对共享资源的访问会被落实为全局变量的访问，是需要通过原子操作来保障其安全性的。而某段代码，在某一进程执行时，不允许其他进程进入，这段代码就称为临界区。

注 1：该全局变量是基于 Linux OS而言的。对于单个进程中的全局变量的同步访问问题，则是线程间资源共享的范畴。

注 2：原子操作，就是指程序中对一变量操作，通过硬件保证，不会被其他执行流插入而打断，就像原子不可分一样。

这些同步的问题不仅在不同的内核控制路径中会有发生，在共享着一般数据的进程间也是存在的。下面就先介绍几个同步内核控制路径的技术。

（ 1）内核禁止抢占

为了简单地解决同步问题，禁止内核的可抢占性：当进程执行在内核模式，其不能被挂起，不允许切换别的进程来运行。这样，在单处理器的系统，内核访问这些数据结构就是安全的。

当然，在内核模式的进程可以自主地放弃 CPU，在这种情况下，其必须保证保存好所有相关数据结构，以便它们保持一致性。这样，当它返回执行时，所用的数据结构是一致的。

这种同步技术对系统的实时响应是有负面影响的。对于 Linux这种可抢占式内核，在访问这些相关的临界区时，只有寻求其他同步技术。而且对于多处理器系统，这种不可抢占技术则根本不能起作用了。为什么？读者不妨自己想想。

（ 2）禁止中断

对于单处理器系统，另一个同步机制是：在进入临界区之前，禁止所有的硬件中断；而在离开临界区时，再使能这些中断。这种机制能够有效，是因为进程的切换是离不开中断的。调度器的运行是与定时器这个中断紧密不可分的。可以说，禁止了中断就禁止了进程的调度。但这个机制，除了简单，离实际使用还很远。如果临界区很大，则会导致硬件中断很长时间也得不到响应。

另外，对于多处理器系统，这招也是不管用的。有的人也许会想，可以同时禁止所有处理器的中断，来实现临界区的排它访问。但与禁止抢占策略一样，会导致实时响应受损。所以，对于实时 Linux，这种同步技术很少用。

（ 3）信号量

信号量（ semaphore）是一个被广泛采用的同步机制，对单处理器系统与多处理器系统都有效。 semaphore就是关联数据结构的简单的计数器。在访问该数据结构之前，所有的内核线程都会先去检查相关的semaphore。每个 semaphore实例可以当作如下对象：

- 一个整型的变量。
- 一个等待进程列表。

• 两个原子性方法：down（），up（）。

down（）方法会对 semaphore的值做递减操作。如果递减后的新值小于 0，则该方法会将运行的进程加入该 semaphore的阻塞队列中。而 up（）方法会对其值做递增操作，如果该值变得大于或等于 0，则会激活阻塞在该 semaphore上的进程。

每个要保护的数据结构都会有它自己的 semaphore，其值初始为 1。当内核控制路径想要访问该数据结构时，它就要先执行 down（）操作。拥有 semaphore的进程在使用完该数据结构之后，应该调用 up（）以释放该 semaphore。

（ 4）自旋锁

在多处理器系统中， semaphore并不总是解决同步问题的最好方法。有些内核数据是必须禁止运行在不同 CPU上的内核控制路径同时访问的。这种情况下，如果为更新该数据结构所需要的时间比较短，则用 semaphore的效率就比较低。为了检查一个 semaphore，内核必须将进程插入相应的 semaphore列表中，然后将它挂起。因为这两个操作都是比较花时间的，在这两个操作期间，有可能其他 CPU将 semaphore早已释放了。

在这种情况下，多处理器系统更愿意使用自旋锁。自旋锁与 semaphore很相似，但它没有进程列表；当一个进程发现该锁被另一个进程锁住了，它就在那里不停地自旋，等待该锁被打开。

当然，自旋锁对于单处理器系统是没有用的。当一内核控制路径想要访问被锁住的数据结构时，其会进入无止境的循环。这样其他内核控制路径就得不到运行的机会，从而导致系统就一直无聊地等待。

（ 5）避免死锁

当内核锁的数目太多时，死锁的问题就会变得特别突出了。在这种情况下，要完全地避免死锁是不可能的。有些 OS，包括 Linux，是通过按预先定义好的顺序来请求锁的方法来避免死锁的。

5.信号与进程间通信

Linux内核提供了向进程通报系统事件的一种机制。每个事件都会有它自己的信号，该信号通常用符号常量（如 SIGTERM）来表示。有以下两类系统事件：

• 异步通报。例如，用户可以发送中断信号SIGINT到前台进程，如按下了中断键（Ctrl+c）。

• 同步通报。内核发送信号SIGSEGV到一进程，如进程访问了一个无效的地址。

POSIX标准定义了大概 20种不同的信号，其中两种由用户定义，可用作用户模式进程间通信和同步的原始机制。总之，进程对信号有两种响应态度：

• 忽略该信号。也就是对信号不做任何处理。

• 异步地执行一特定的例程（我们常将该例程称为信号句柄）。

如果进程不指定上述的方法，内核则会根据信号做默认动作。下面是5个可能的默认动作：

1）忽略该信号。

2）终止进程。

3）写执行上下文和地址空间等相关内容到一个文件（ core dump），然后终止进程。

4）挂起进程。

5）如果该进程被停止了，则重新启动该进程执行。

内核的信号处理是相当精巧的，因为 POSIX语法允许进程短暂地阻塞信号，这样，接收该信号的进程可以暂时不响应该信号。另外，SIGKILL和 SIGSTOP两个信号不能直接被进程处理或被忽略。

注：我们要提醒读者注意，阻塞信号与前面的忽略信号是两种不同的信号响应模式；阻塞信号其实是异步处理中的一种处理方式。

Linux OS还引进了 System V IPC，用于用户模式进程间的通信包括semaphore、消息队列和共享内存。

内核实现了这些装配件作为 IPC的资源。进程通过调用 shmget（）、semget（）或 msgget（）系统调用得到相关的资源。像文件一样，IPC资源是永久性存在的：它们必须由创建进程、当前拥有者或超级用户进程显性地释放。

semaphore与“ 4.同步与临界区”小节中描述的 semaphore相似，当然其用于用户模式下的进程，是由拥有者进程运行时在系统中创建；而“ 4.同步与临界区”小节中 semaphore是由 Linux OS内核启动时创建，并用来保障内核相应资源可重入的。而消息队列则可以让进程通过 msgsnd（）和 msgrcv（）系统调用来交换消息，这两个系统调用分别将消息插入特定的消息队列以及从该消息队列抽取消息。

共享内存向进程间提供了最快的相互交换共享数据的方法。进程通过发出 shmget（）系统调用来创建一个指定大小的共享内存。在获得IPC资源 ID之后，该进程会调用 shmat（）系统调用，得到该共享内存在本进程空间中的起始地址。进程可以调用 shmdt（）系统调用将该共享内存从进程空间删除。共享内存的实现依赖于内核如何实现进程地址空间。

进程间的通信是基于共享资源的同步技术的，其目的是要利用同步技术实现进程间信息的交流。

6.进程的生死

Linux OS用 fork（）系统调用来创建一个新的进程，用 _exit（）系统调用终止进程，而 exec（）之类的系统调用则用来装载新的程序。

调用 fork（）的进程是父进程，而新生成的进程则称为子进程。父进程和子进程可以相互发现，因为描述每个进程的数据结构中都会有一个指针指向其直接的父进程，还有指针指向它所拥有的所有直接的子进程。

fork（）简单实现会要求父进程的数据和父进程的代码可以被复制，并且这些备份会被指派给子进程。这样的话就会特别消耗时间。当前的 Linux内核一般依赖于硬件分页单元，使用了 Copy-On-Write方

法，以推迟页复制（只有当父进程或子进程需要对该页进行写操作时，才会去做复制动作），这样将极大地提高这方面的性能。

_exit（）系统调用会去终止该进程。内核会通过释放该进程所拥有的资源，以及向父进程发送 SIGCHLD信号，以对该系统调用进行响应处理。 SIGCHLD信号会默认为父进程忽略。

（ 1） zombie进程

父进程如何询问其子进程的终止状况？ wait4（）系统调用让一个进程可以等待其某一子进程的终止；它会返回被终止子进程的 PID。

当执行该系统调用时，内核会检查其某一子进程是否已被终止。这样一个特定的 zombie进程状态被引进来，以代表终止了的进程：某一进程会一直保持在该状态，直到它的父进程执行 wait4（）系统调用。该系统调用句柄会从进程描述符的域中抽取关于资源使用情况的数据；在这些数据被收集好后，该进程描述符就会被释放。如果在 wait4（）系统调用被执行时，没有子进程被终止，内核就会让该进程进入等待状态，直到有子进程终止为止。

Linux内核也实现了 waitpid（）系统调用，它会允许某进程等待某特定子进程的终止。

父进程在退出前调用 wait4（），保留好子进程的相关信息是一个好的经验。因为如果在父进程没有发出该调用之前就终止了，这些 zombie进程就会一直占用许多有用的内存。

对这个问题， Linux采用了一个变通的解决方案，它依赖于某特定的系统进程，该进程被称作 init，该进程在系统初始化的时候被创建。当一个进程终止时，内核会将其子进程的父进程变更为 init，进而让 init单独去关注 zombie进程。

（ 2）进程组和 login会话

Linux OS引进了进程组，以便代表 "job"的抽象。例如以下的命令行：

```
$ls | sort | more
```

Shell就是支持进程组的，如 bash，会为上面 3个进程创建一个新的进程组。这样 Shell就会把这 3个进程当作一个实体来处理。每个进程描述符都会有一个域来装载进程组 ID。而每个进程组也会有一个组长，该组长的 PID与进程组 ID一致。一个新创建的进程初始是被插入其父进程所在的进程组的。

Linux内核也引进了 login会话，一个 login会话中包含了基于一特定终端进程的子孙的所有进程。在一进程组中所有进程必须在同样的 login会话中。但一个 login会话，在同一时间可以有多个进程组；但只会有一个进程组在前台。当后台进程想要访问终端时，它会接收到 SIGIN或 SIGOUT信号。 Shell中内部命令 bg与 fg可以让进程组到后台或前台。

3.2.2 内存管理综述

内存管理是 Linux内核中最复杂的活动，这里将描述 Linux是如何进行内存管理的，以及提出在内存管理中会碰到的问题。

1.虚拟内存

虚拟内存是应用程序用到的逻辑存储空间，其会用到 MMU（内存管理单元）。虚拟内存要完成以下目标，并有诸多优点：

- 多个进程得以同时运行。
- 运行的应用程序所允许访问的内存空间可以大于设备物理的内存地址空间。
- 进程执行的程序代码只有部分装载进了内存。
- 每个进程被允许访问的是可用物理内存的一个子集。
- 进程可以共享库或程序的单一内存镜像。
- 程序是可以重寻址的，即可以被放在物理内存的任何地方。

正是因为虚拟内存，程序员所编写的代码是独立于机器的，因为不用考虑物理内存的组织。

当进程使用的是虚拟地址时，内核与 MMU就要协作，以便为这些进程寻找对应的物理地址。

现代 CPU都会有 MMU硬件电路，以自动地把虚拟地址翻译成物理地址。并且，可用的 RAM都以页为单位分成了一块块的，这些页一般是4KB或 8KB大小。此时就引入页表，以便让虚拟地址与物理地址对应起来。

注：我们也常将 Linux中的 swap称作虚拟内存，但它是将硬盘空间的一块用作内存。主要用于在 RAM空间不够用时，将 RAM中不活跃或不重要的数据，先放置到这块硬盘空间上；当需要这些数据时，再直接转换到 RAM中，这样可以减少相关文件系统的操作。显然，这个虚拟内存与这里讲的虚拟内存是两个概念，请读者千万不要混淆，具体要根据上下文来理解。

2.RAM的使用

Linux OS都把 RAM清晰地分成两部分。其中几兆的空间被专门划给内核 Image使用（包括内核的代码与数据）。 RAM余下的部分则被虚拟内存所掌控，该部分的使用有 3种方式：

- 满足内核关于buffer＼描述符＼其他动态内核数据结构的请求。
- 满足进程关于一般内存区和文件映射的请求。
- 满足磁盘性能或其他要缓存的设备性能所做Cache的请求。

上面每种请求类型都是有意义的。但是，可用的 RAM是有限的，在这些请求类型间就要做些取舍，特别当可用内存所剩无几时，这种权衡就更有必要了。另外，当可用内存到达某些关键的门限时， Linux内核会调用 page-frame-reclaiming算法，以便释放出更多的内存。哪些页帧（ page frames）是最适合回收（ reclaiming）的？这个问题不好回答，也没有好的理论可以支持这个问题。唯一可用的解决办法是：小心地根据经验调节相应的算法。

除了要不断回收 RAM的页空间，还有一个关键的问题必须解决：整理内存碎片。当 free page frames太小时，对内存的请求就可能失败。内核通常会强制要求使用物理连续的内存区域，这样对内存的访问就会有更少的跳转动作，有利于提高程序的执行效率。事实上，即使内存空闲总量足够大时，如果碎片太多，申请内存也还是有可能失败的。

3.内核内存分配器

Linux中的 KMA（ Kernel Memory Allocator，内核内存分配器）是一个子系统，用来满足上述对内存区的前两类请求（关于动态内核数据结构的请求与关于一般内存和文件映射的请求）。这些请求部分来自内核子系统，部分则来自用户请求发出的相关系统调用。一个好的KMA有以下特性：

• 反应快。事实上，这是最关键的属性，因为它可以被所有的内核子系统调用（包括中断处理例程）。

• 应尽可能地减少对内存的浪费。

• 应努力减少内存碎片。

• 可与其他内存管理子系统（如Cache管理子系统）协作，以便从它们那里借到与释放页帧。

现代 KMA常会基于以下技术：

• 资源映射分配器。

• Mckusick-Karels分配器。

• Buddy系统。

• Mach’s Zone分配器。

• Dynix分配器。

• Solaris’s Slab分配器。

Linux的 KMA用的是基于 Buddy系统的 Slab分配器。

4.进程虚拟地址空间处理

进程的地址空间包含了该进程可以引用的所有虚拟内存地址， Linux内核通常将进程虚拟地址空间装载为一个内存地址描述符的列表。所谓列表，是指一个进程会有多个内存地址描述符，这些描述符被统一放置在一张表中，由这张表来描述整个进程的虚拟地址空间。例如，当某进程通过诸如 exec（）类的系统调用来启动某些程序的执行时，内核会为新启动进程的虚拟地址空间做相应分派工作。在该空间中会包含以下内存区域：

- 该程序可执行的代码。
- 该程序初始化了的数据。
- 该程序没有初始化的数据。
- 初始的程序栈（即用户模式栈）。
- 共享库的可执行代码与数据。
- 堆（为该程序动态申请的内存）。

所有的 Linux OS都采取了一种内存分配机制：按需分页。通过该机制，某一个进程可以在该程序在物理内存中没有任何页面的情况下，就启动其程序的执行。当它访问一个不在物理内存的页时， MMU就会产生异常；该异常句柄会找到受影响的内存区域，分配一个空页，并将该页初始化合适的数据。同样，当该进程通过 malloc（）或 brk（）系统调用动态地请求内存时， Linux内核会更新该进程堆（heap）内存域的尺寸。因此，当进程通过引用其虚拟内存地址而产生异常时，某一页帧会被分派给该进程。

虚拟地址空间还允许使用其他有效的机制，例如前面提到的 Copy-On-Write机制，以共享父进程已有的代码空间与数据空间。例如，当新的进程被内核创建了， Linux内核只是分配父页帧到子进程地址空间，但会标记这些空间是只读的。我们可以把这些分配与标记理解为只是设置与修改上述的进程内存地址描述符而已。当子进程或父进程想要

修改这些页中的内容时，就会产生异常。该异常处理例程就会为受影响的进程分配一个新的页，并用原始页中的内容初始化该新分配的页，最后再在该新分配的页上做想要的修改。

5.Caching

Caching即把物理内存的一块用作磁盘或其他块设备的 Cache。因为磁盘驱动很慢，因此在系统性能中磁盘常是一个瓶颈。事实上，在嵌入式设备中，一般使用 Flash取代磁盘。但目前所用的 Nand Flash，相对 RAM的访问也还是太慢了。因此 Linux引进一个策略：尽可能地延迟写磁盘操作。结果，先前来自磁盘的数据，即使不再为任何进程使用，也会尽可能继续待在 RAM中。

但当程序对 Cache中数据进行访问，尤其是写操作时，可能会导致 Cache中数据与硬盘或块设备中对应数据不一致的情况；当 Cache中数据与永久保存在存储介质上的文件或数据不一致时，就称该 Cache 为“脏 buffer”。 Sync（）系统调用用来强制所有的“脏 buffer”要被写到磁盘中，以保持 Cache与硬盘等设备上数据的一致性。为了避免数据损失，所有的 OS都会周期性地关注该 Cache，并将“脏 buffer”写回到磁盘上。

3.2.3　文件系统综述

Linux OS的设计是以文件系统为中心的。

1.文件

Linux是一个信息容器，用来承载结构化的字节； Linux内核并不解释文件的内容。许多程序库实现了更高层抽象，例如数据库系统会用 field来结构化记录，并通过 key来寻址记录。但是在这些库的程序必须依赖内核所提供的文件系统调用。从用户观点来看，文件都是按树形结构的命名空间来组织的，如图 3-4所示。

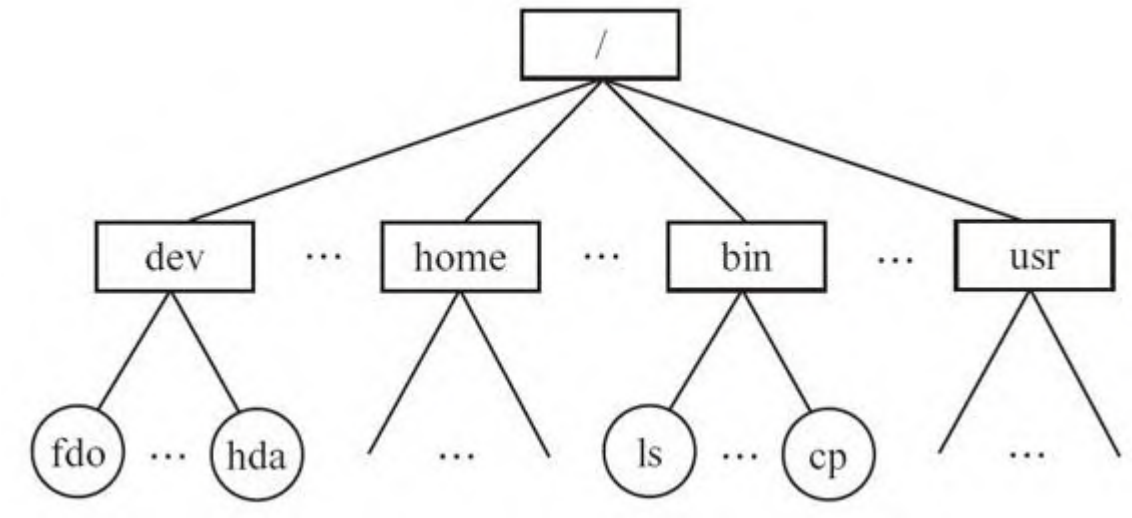


图3-4　Linux树形文件系统

所有节点（不包括叶子），都称为目录。目录节点包含了其下目录与文件的信息。文件或目录的名称由一组专有的 ASCII字符组成，这些字符当中不能有“ /”和 null字符 "\0"。大多数的文件系统会对文件名的长度进行限制，一般不允许超过 255个字符。对应于文件树的根目录，被称作 root目录。通常， root用“ /”表示。在同一目录内的文件名不能相同。

Linux会为每个进程关联一个当前工作目录；当前工作目录是属于进程的执行上下文的，它用来标识该进程正在使用的目录。为了标识特定的文件，进程使用 pathname。如果 pathname的第一个字符是“ /”，则该路径是绝对路径；否则就称作相对路径，该路径的起点是当前目录。

为了指定文件名，“ .”与“ ..”也会被使用，其中前者表示当前目录，而后者则是父目录。特别的，如果当前工作目录是 root目录，则两者都指 root。

2.硬链接与软链接

Linux通过链接来为文件向用户程序提供多个入口。 Linux链接有两种：一种称为硬链接（ Hard Link），另一种称为符号链接（ Symbolic Link），也常被称为软链接。默认情况下， In命令产生硬链接。

硬链接通过索引节点（ inode）进行链接。这样多个文件名指向同一索引节点，也就使得一个文件可以拥有多个有效路径名；此时只删除一个链接，并不会影响到该索引节点，只有当最后一个链接被删除时，文件的数据块及对应的索引节点对象才会被真正删除或释放。但硬链接有以下限制：

- 不能对目录创建硬链接。
- 硬链接只能针对同样的文件系统来创建。

为了克服这些限制，又引入了软链接。这个符号链接其实是短文件，其包含了另一文件专有的路径名。该路径名可以是文件，也可以是目录。这些目录与文件可以在任何文件系统中，甚至可以是一个根本不存在的文件。

注：硬链接就是创建一个新的文件，而软链接只是一个符号上的链接。其中软链接的修改都是直接针对所链接的文件的。硬链接牵涉到文件系统的 inode，而软链接则不牵涉到 inode。当然，对硬链接文件 inode所指向的文件内容是被链接文件的内容；从这一点我们也好理解为什么硬链接不支持目录：因为目录的 inode结构中已包含了该目录的所有内容，而没有另外的存储空间存储着它的内容。

引入这些链接文件类型，主要就是为了可以让程序使用当前工作目录这个属性，这有利于提高程序的效率。

3.文件类型

Linux文件可以是以下类型中的一种：

- 常规文件。
- 目录。
- 软链接。
- 面向块的设备文件。
- 面向字符的设备文件。

• 管道与命名管道（也称为FIFO）。

• 套接口。

前 3种文件类型，对于所有的 Linux文件系统都是一致的。

设备文件与 I/O设备相关，也与集成到内核的设备驱动相关。例如，当一个程序访问一个设备文件时，它将导致对与该文件相关 I/O设备文件的操作。

管道与套接口是特殊的文件，用于进程间的通信（在前面我们讲过信号、 semaphores等更直接的进程间通信方式，我们会在后面对它们进行更详细的讨论，以便读者更清楚地理解这些通信方式的应用场景）。

4.文件描述符与inode

Linux对文件的内容与文件的相关信息之间有一个清晰的区分。除了设备文件与特殊文件系统的文件之外，每个文件都是由一串字节组成的。该文件不包括任何控制信息，如文件的长度或 EOF定界符。

所有用于文件系统处理文件所需要的信息都被包含在一个数据结构中，该数据结构叫做 inode。每个文件都有它自己的 inode，该inode将被文件系统用来识别文件。

Linux文件系统的 inode提供了以下属性（这些属性是为 POSIX标准所要求的）：

• 文件类型。

• 与该文件关联的硬链接的数目。

• 文件的长度（以字节为单位）。

• 设备ID（即装载该文件的设备的ID）。

• inode号，其用于文件系统中识别该文件。

• 文件拥有者的UID。

• 该文件的用户组号。

• 几个时间戳，指定inode状态改变的时间、最后访问的时间、最后修改的时间。

• 访问权限与文件模式（参见下一小节）。

关于这些 inode信息如何在 Linux文件系统中起作用，以及文件系统如何定位文件并取得其内容的机制，会在后面有更详细的讲解。

5.访问权限与文件模式

一个文件的潜在用户有 3种类型：

• 该文件属于该用户。

• 与文件拥有者同一组的组员。

• 其他用户。

对文件的访问有 3种权限：读、写与执行，针对这 3类用户将有独立的 3种权限的设置。

另外，还有 3个标志（ flag）给可执行文件： suid（ Set User ID）、 sgid（ Set Group ID）以及 sticky。这几个 flag用于定义文件的模式。

（ 1） suid

进程执行时，对应可执行文件将保存着该进程拥有者的用户 ID（ UID）。但是如果该可执行文件将 suid flag置位了，进程 UID则会变为该文件拥有者的 UID。

（ 2） sgid

进程正执行一个文件时会保持原有进程组为其用户组 ID。但是，如果可执行文件将 sgid flag置位了，该进程 GID就会变为被执行文件的 GID。

（ 3） sticky

可执行文件拥有 sticky flag置位，其用来向内核请求，在其终止运行时，还保持该程序在内存。

注： sticky flag其实已经废弃了，因为现在采用的是代码页共享的技术，没有哪个页可以常驻内存。

若一个文件被某个进程创建，则该文件的 UID就是该进程的 UID。其GID可以是该进程的 GID，也可以是该文件所处父目录的 GID，具体依赖于父目录的 sgid flag：如果该标志置位了，则该文件的 GID就是父目录的 GID；否则就会新建一个 GID。

6.文件处理系统调用

当用户不管是访问一般文件或是目录时，他事实上访问的是硬件块设备上的某些数据。这样，文件系统就是关于硬件块设备分区的用户级的视角。因为在用户模式下的进程不能直接与低级硬件组件交互，因此对文件的实际操作都应在内核模式中完成。所以， Linux OS定义了几个相关系统调用，以便处理文件。

Linux内核花费了很大的精力于硬件块设备功能驱动及该类设备访问控制上，以得到良好的整体系统性能。在下面的章节中，我们将描述Linux是如何处理文件的，以及内核如何响应相关的文件系统调用。

（ 1）打开文件

进程只能访问已打开了的文件。为了打开文件，进程要调用下面的系统调用：

```
fd=open（path，flag，mode）；
```

---

path：指定文件的路径名。

flag：指定该文件将如何打开（是只读，还是可读可写，以及写是否为 append），它也可以指定一个不存在的文件是否要被重新创建。

mode：指定一个新建文件的访问权限。

该系统调用会创建一个 "open file"对象，并返回叫做文件描述符的ID。该 open file对象包含了：

1）一些文件处理的数据结构，例如一组 flag指定该文件是按何种模式打开的。 offset域指定了该文件当前所要访问处理的起始位置等。

2）一些内核函数的指针，以便该进程可以调用这些函数。这组所许可的函数与上面系统调用中的 flag参数紧密相关。

我们将在后面更详细地讨论 open file对象。下面我们主要关注两点：

1）文件描述符代表了进程与 open file之间的交互，其中 open file对象中包含了与交互相关的数据。同样的 open file对象可以为同一进程中的多个文件描述符所识别。

2）几个进程可以同时打开同一文件。在这种情况下，文件系统会为每个文件指派不同的文件描述符，同时会指派不同的 open file对象。当这种情况发生时， Linux文件系统不会提供针对这些进程对同一文件所发出的 I/O操作进行同步处理。但是有几个系统调用，如flock（），以便允许进程们同步它们的访问。

为了创建新文件，可以调用系统调用 creat（），它的处理与open（）是一样的。

（ 2）访问已打开的文件

常规的 Linux文件可以顺序寻址，也可被随机寻址；其中设备文件与命名管道（ pipe）通常都被顺序访问。在这两种访问方式中，内核将文件指针存储到 open file对象中，该指针会指定下一个要访问的地址。

顺序访问隐含着以下假定： read（）和 write（）系统调用总是引用当前文件指针所指定的位置。为了修改该值，程序必须显性地调用lseek（）系统调用。当一个文件被打开时，内核会设置文件指针到该文件的第一个字节。

该 lseek（）系统调用会要求下面的参数：

```
newoffset=lseek（fd，offset，whence）；
```

fd：表明该打开文件的描述符。

offset：指定一个有符号整数，以计算出新的文件指针的位置。

whence：指定新的访问位置是由 offset值 +当前指针，还是由最后一字节的位置 +offset值得到。

read（）系统调用则要求下面的参数：

```
nread=read（fd，buf，count）；
```

fd：文件描述符。

buf：指定在进程地址空间的 buffer地址，数据将会被传送到那里。

count：指定要读取的字节数。

当处理此类系统调用时，内核会试着读取 count个字节。 write（）的参数是相似的。

（ 3）关闭文件

```
res=close（fd）；
```

（ 4）重命名文件与删除文件

为了重命名文件或删除一个文件，进程并不需要打开该文件。事实上，这两个操作并不会影响到所操作文件的内容，但会影响到相关目

录文件的内容。例如：

```
res=rename（oldpath，newpath）；
```

该系统调用将变更某文件名。而

```
res=unlink（pathname）；
```

则删除了某文件的某链接，相应地也会删除相关目录中的该链接文件条目；当文件的 Link数目值为 0时，该系统调用将会删除源文件本身。

3.2.4　设备驱动简述

设备驱动开发是开发过程中经常要面临的任务，这也是本书的主题。

Linux内核是通过设备驱动与 I/O设备交互的。设备驱动被包含在Linux内核中，由控制一个或多个设备的数据结构与功能函数构成，如硬盘、键盘等。每个驱动会通过一个特定的接口，与内核的其他部分（甚至是其他驱动）交互。这个方法有以下好处：

- 特定于设备的代码，可以被封装在特定的模块中。
- 厂商可以在不知道内核源码的情况下，增加新的设备，只要知道特定的接口即可。
- 内核会用统一的方式来处理所有设备，并通过同样的接口来访问这些设备。
- 可以以模块的方式来写驱动，并可以动态地加载这些模块，也可动态地卸载它们。

图 3-5显示了设备驱动与内核其他部分以及进程的交互。

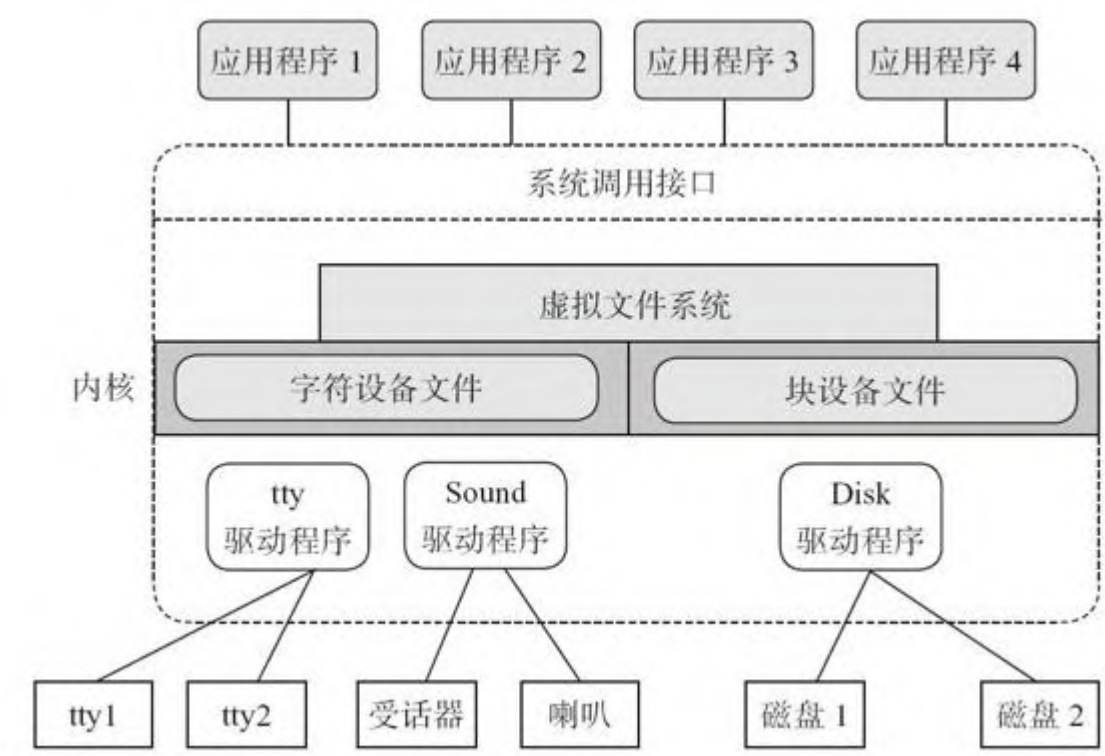


图3-5　设备驱动接口

一些用户程序会想要操作硬件设备，它们会向内核发出请求，请求访问 /dev目录下的文件。事实上，设备文件就是用户可见的设备驱动的接口。每个设备文件都会引用到特定的设备驱动，它会被内核调用，以完成对相应硬件组件的操作。

第4章　Linux内核编程与内核模块

如第3章所讲的，Linux内核各组成部分是由一个个模块来实现的。本书所关注的设备驱动，也是由一个或多个模块来实现的。

本章将讲述Linux内核编程，以及内核模块的基本代码骨架。事实上，从本章开始我们才真正接触到代码。敬请读者认真阅读下面的样例代码，最好是找个环境验证相应的样例代码，相信你将因此受益颇多。

4.1 Linux内核源代码目录结构

为编写 Linux内核代码，我们应先读读源代码，了解 Linux内核源代码的目录结构。本书的样例代码是基于 Linux 2.6.35的。主要包含下面的目录：

1） arch：该目录包含与 CPU硬件系统结构相关的代码。每个 CPU系列都独自占有一个目录，如 ARM、 MIPS、 AVR32、 x86、 ia64等。

2） block：该目录包含块设备驱动程序中进行 I/O调度的功能代码。

3） crypto：该目录包含加密 /解密算法，以及压缩和校验等功能代码。

4） documentation：该部分是一些文档，在该文档中对内核的各个部分进行了一般性的阐述。在承担具体的开发任务或进行相关模块的研发时，建议到这个目录下找相应的文档读读，相信会有所收获的。

5） drivers：该目录包含各设备程序的功能代码。每种类型的设备驱动常占有一个独立的子目录，如 char、 block、 net、 input、 power等。

6） fs：该目录包含 Linux内核所支持的各种文件系统，如 ext、 jffs2、 yaffs2、 fat、 ntfs等。

7） include：该目录包含一些头文件，其中与 Linux系统相关的头文件就放置在该目录下的 linux子目录中。

8） init：该目录包含 Linux内核的初始化功能代码。

9） ipc：该目录包含进程间通信的功能代码。

10） kernel：该目录包含进程调度、定时器等功能代码。

11） lib：该目录包含库或用于生成库的代码。

12） mm：该目录包含内存管理功能代码。

13） net：该目录包含网络相关的功能代码，其实现了各种常见的网络协议。

14） scripts：该目录包含一些脚本文件，用于配置内核。

15） security：该目录包含 Linux安全管理方面的代码，如账号等。

16） sound：该目录包含 ALSA、 OSS音频子系统的核心代码，以及一些常用的音频驱动。

17） usr：该目录包含实现 cpio工具的功能代码。

不同平台的 Linux源代码目录可能会稍有差别，但主要的就是上面所列的。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

4.2 Linux内核的编译与启动

下面基于 2.3节所搭的开发环境，针对一款 ARM平台的 Linux 2.3.35内核，进行编译，并下载与启动新内核 Image的过程。这个过程对于所有的平台与 Linux 2.6内核是类似的。

1）获得对应平台的 Linux 2.6.35的内核源码，将这部分源码放置在2.2节所述 Android源码下的 kernel目录下。

2）切换到 kernel目录下。比如，工程目录是/home/projects/test/android/kernel，则执行以下命令：

```
$cd/home/projects/test/android/kernel
```

3）执行 make mrproper命令，以确保 kernel的源代码目录下没有不正确的 .o等文件，并且该目录下各文件的依赖性是完整的。命令如下：

```
$make mrproper
```

4）执行 make menuconfig命令，以配置内核，对组成 Linux内核的模块进行增减。命令及界面如图 4-1所示：

```
$make menuconfig
```

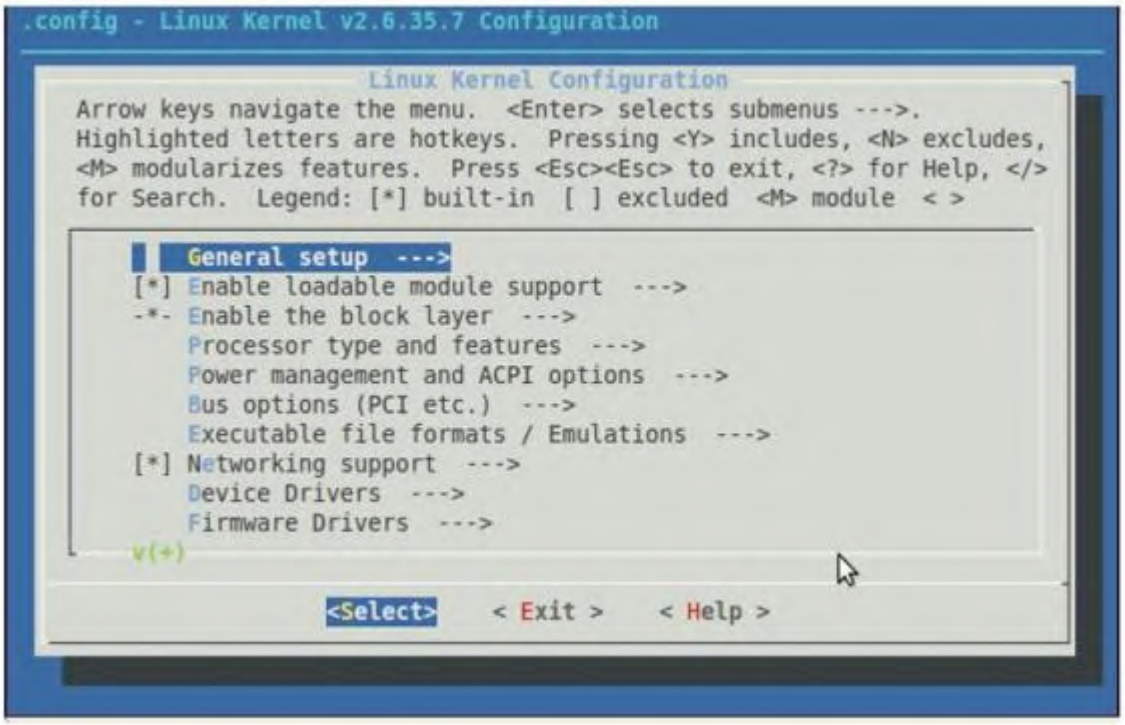


图4-1 Linux内核编译配置

5）在完成上面的修改后退出 menu时，提示是否要保存修改。选择 yes，以修改同目录下的 .config文件。

6）执行 make uImage命令，以编译内核。命令如下：

```
$make uImage
```

这里有两点还需要说明。一是交叉编译环境；二是 Linux的编译配置脚本。

由于编译环境所用的 PC往往是 x86等 CPU平台，与目标机 ARM等平台不一样。为了在 x86上编译出在 ARM平台上运行的 Image，要用到交叉编译工具。该工具一般由目标机 CPU平台厂商提供。我们就用了 Android自带的 ARM交叉编译 toolchain arm-eabi-4.4.3，该工具在 Android目录下： /prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-xxx。

Linux各模块的配置文件是 Kconfig文件。通过该文件，可以把要增加的模块配置添加到 Linux配置选项里，这样就可以通过 make menuconfig，选择是否要配置相应的模块。仅是配置选上了，模块也不一定会编译。 Linux模块是否编译则由 Makefile控制。而 Kconfig与 Makefile具体是如何控制 Linux模块的配置与编译的，在后面的样例模块将有更进一步的讲述，大家照葫芦画瓢即可。更具体的，大家可以参考 Linux内核的 document文档。

最后一点要说明的，很多目标板要求生成的内核 Image进行压缩、更名等处理。这时，最好修改 Linux kernel目录下 Makefile文件，将相应的处理命令加入该脚本文件，将极大地方便 Linux内核的编译。

正常的话，内核编译后会得到 zImage等内核 Image。可以将编译好的 Kernel Image烧写在目标机器里。一旦该 Linux内核得以运行，那目标机器就将完全由该 Linux OS控制。要运行 Linux内核，必须想办法将 Linux内核 Image从 Nor/Nand Flash/Sd卡等非易失性存储体加载到目标机器的 RAM中。这部分工作需要另一个组件 bootloader来辅助。当然，实现该 bootloader的有很多种。在 Android中，大多是基于开源的 Uboot。一般平台厂商会基于开源的 Uboot和自己 CPU平台的特性，做相应的修改与扩展。事实上， bootloader里除了加载与启动 Linux内核 Image外，还会增加诸如开机模式选择等功能。 Uboot不是本书的重点，这里就不做更多的阐述。

4.3 Linux内核的 C编程

虽然前面我们讲过， Linux引进了面向对象的思想，但为了效率，其所用的编程语言主要是 C。其最底层甚至用的是汇编语言。

Linux C编程的习惯与 Windows下的匈牙利法中以首字母区分变量或函数名不同，其多在两个单词之间，以下划线隔开，如下：

```
int min_value, max_value;
void receive_value(int my_value);
```

另外， Linux的编译使用的是 GNU C，而不是标准的 ANSI C。 GNU C对标准 C做了增强与扩展，如下：

1）允许长度为 0的数组。

GNU C允许使用零长度数组，在定义变长对象的头结构时，这个特性非常有用。例如：

```
struct linear_array
{
    int length;
    char data[0];
};
```

char data[0]就意味除了为结构体 linear_array的实例分配内存外，并不会为数组 data[]分配内存，因此 sizeof（ struct linear_array） =sizeof（ int）。而在程序需要的时候，可再为数组 data[]申请内存。

2）支持 case x...y语句，以响应满足区间 [x， y]的值的分支条件。这给代码的编写带来了方便。如下：

```
char c;
switch (char_x)
{
    case '0'... '9': c -= '0';
        break;
    case 'a'... 'f': c -= 'a' - 10;
        break;
    case 'A'... 'F': c -= 'A' - 10;
        break;
}
```

代码中的 case'0'...'9'等价于标准 C中的如下代码：

```
case '0': case '1': case '2': case '3': case '4': case '5':
case '6': case '7': case '8': case '9':
```

3）引入了语句表达式。

GNU C把包含在括号中的复合语句看作一个表达式，称为语句表达式，它可以出现在任何允许是表达式的地方。可以在语句表达式中使用原本只能在复合语句中使用的循环变量、局部变量等，例如：

```
#define min_var(type,x,y) \
({ type _ _x = (x); type _ _y = (y); _ _x < _ _y ? _ _x: _ _y;
})
int inta, intb, minint;
float floata, floatb, minfloat;
minint = min_var(int, inta, intb);
minfloat = min_var(float, floata, floatb);
```

因为重新定义了 __x和 __y这两个局部变量，所以以上述方式定义的宏将不会有副作用。而在标准 C中，对应的如下宏则会产生副作用：

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

代码 min（ ++ia， ++ib）会被展开为（（ ++ia）＜（ ++ib）（ ++ia） :（ ++ib）），传入宏的参数被增加两次。而在 GNU C中引入语句表达式，通过重新定义变量的方式很好地规避了这个问题。

4）引入了 typeof关键字。

该关键字其实是一个运算符，用于求得某变量的类型。有了这个关键字之后，前面定义的宏 min_var（ type， x， y），就不再需要传入type值了。如下：

```
#define min_var(x,y) ({ \
const typeof(x) _x = (x); \
const typeof(y) _y = (y); \
(void) (&_x == &_y); \
_x < _y ? _x : _y; })
```

代码行（ void）（ &_x==&_y）用来检查 _x和 _y的类型是否一致。如果不一致，将在程序编译时报指针类型不一致的错误。

5）扩展宏的参数也可变。

在 GNU C中，不仅函数的参数可以变，而且宏的参数也可以变。这里的可变是指，参数的个数与类型可以不固定。标准 C中的 printf函数就是这种情况，其中的变量将随着 format的不同而有变化，如下：

```
int printf( const char *format [, argument]... );
```

而在 GNU C中，宏也可以接受可变数目的参数，例如：

```
#define pr_debug(fmt,arg...) \
printk(fmt,##arg)
```

随着 fmt的设定，这里 arg表示其余的参数可以是零个或多个，例如下列代码：

```
pr_debug("%s:%d",filename,line)
```

会被扩展为：

```
printk("%s:%d", filename, line)
```

在上面的宏定义中使用了“ ##”，表示 arg没有任何参数的时候，前面的逗号就变得多余了。使用“ ##”之后， GNU C预处理器会丢弃前面的逗号，这样，代码：

```
pr_debug("success!\n")
```

会被正确地扩展为：

```
printk("success!\n")
```

而不是：

```
printk("success!\n",)
```

6）允许对标号元素进行不按顺序的初始化。

标准 C要求数组或结构体的初始化值必须以固定的顺序出现，在 GNU C中，允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前添加“ [INDEX]=”，当然也可以用“ [FIRST… LAST]=”的形式指定一个范围。例如，下面的代码定义一个数组，并把其中的所有元素赋值为 0：

```
unsigned char array[MAX] = { [0 ... MAX-1] = 0 };
```

下面的代码借助结构体成员名初始化结构体：

```
struct file_operations ext2_file_operations =
{
    llseek: generic_file_llseek,
    read: generic_file_read,
    write: generic_file_write,
    ioctl: ext2_ioctl,
    mmap: generic_file_mmap,
    open: generic_file_open,
    release: ext2_release_file,
    fsync: ext2_sync_file,
};
```

但是，在 Linux 2.6.35中，类似的代码建议尽量采用标准 C的方式，这样会让代码看起来更整洁。如下所示：

```
struct file_operations ext2_file_operations =
{
    .llseek = generic_file_llseek,
    .read = generic_file_read,
    .write = generic_file_write,
    .aio_read = generic_file_aio_read,
    .aio_write = generic_file_aio_write,
    .ioctl = ext2_ioctl,
    .mmap = generic_file_mmap,
    .open = generic_file_open,
    .release = ext2_release_file,
    .fsync = ext2_sync_file,
    .readv = generic_file_readv,
    .writev = generic_file_writev,
```

```
    .sendfile = generic_file_sendfile,
};
```

7）引入两个保存当前函数名的标识符。

这两个标识符分别是 __FUNCTION__和 __PRETTY_FUNCTION__。其中前者用来保存源码中所用的函数名；后者则会根据函数所在语言环境的不同而有所变化。当然，当在 GNU C语言环境下，这两个标识符的值是一样的。例如：

```
void test()
{
    printf("This is function:%s", _ _FUNCTION_ _);
}
```

代码中的 __FUNCTION__意味着字符串 "test"。

8）允许声明函数、变量和类型的特殊属性。

这个增强将有利于进行手工代码优化、定制和代码检查。通过在函数、变量和类型后面添加 __attribute__（（ ATTRIBUTE））。其中ATTRIBUTE为属性说明，如果存在多个属性，则以逗号分隔。 GNU C支持 noreturn、 format、 unused、 aligned、 packed等十多个属性。

noreturn属性作用于函数，表示该函数从不返回。这会让编译器优化代码，并消除不必要的警告信息。例如：

```
# define ATTRIB_NORET _ _attribute_ _((noreturn))
asmlinkage NORET_TYPE void do_exit(long error_code)
ATTRIB_NORET;
```

---

format属性也用于函数，表示该函数使用 printf、 scanf或strftime风格的参数，指定 format属性可以让编译器根据格式串检查

参数类型。例如：

```
asmlinkage int printk(const char * fmt, ...) _ _attribute_ _
((format (printf, 1, 2)));
```

printk函数的第一个参数是格式串，从第二个参数开始都会根据第一个函数所指定的格式串规则检查参数。

unused属性作用于函数和变量，表示该函数或变量可能不会被用到，这个属性可以避免编译器产生警告信息。

aligned属性用于变量、结构体或联合体，指定变量、结构体或联合体的对齐方式，以字节为单位，例如：

```
struct example_struct
{
    char a;
    int b;
    long c;
} _ _attribute_ _((aligned(4)));
```

这个例子的属性限定该结构类型的变量以 4字节对齐。

packed属性作用于变量和类型，用于变量或结构体成员时表示使用最小可能的内存，用于枚举、结构体或联合体类型时表示该类型使用最小的内存。

9）扩展了标准 C的库函数。

GNU C实现了所有标准 C的库函数，如 malloc（）等。另外， GNU C库函数中，还扩展了标准 C所没有的函数， GNU C在这些函数的前面加上 __builtin以标识。如下：

① __builtin_return_address（ LEVEL）：用于告诉将返回当前函数或其调用者的返回地址，参数 LEVEL指定调用栈的级数，如 0表示当

前函数的返回地址， 1表示当前函数的调用者的返回地址。

② __builtin_constant_p（ EXP）：用于判断一个值是否为编译时常数，如果参数 EXP的值是常数，函数返回 1，否则返回 0。

③ __builtin_expect（ EXP， C）：用于为编译器提供分支预测信息，其返回值是整数表达式 EXP的值， C的值必须是编译时常数。

Linux内核代码中，经常会出现 do{}while（ 0）这样的语句。许多人感到不解：认为 do{}while（ 0）毫无意义，因为它只会执行一次，加不加 do{}while（ 0）效果是完全一样的。其实， do{}while（ 0）主要用于宏定义中。例如：

```
#define SAFE_FREE(p) do{ free(p); p = NULL;} while(0)
```

假设这里去掉 do{}while（ 0），即定义 SAFE_FREE为：

```
#define SAFE_FREE(p) free(p); p = NULL;
```

那么以下代码：

```
if(NULL != p)
    SAFE_FREE(p)
else
    ...//do something
```

会被展开为：

```
if(NULL != p)
    free(p); p = NULL;
```

```
else
    ...         //do something
```

展开的代码中就出现了问题： if分支后有两个语句，导致 else分支没有对应的 if，编译失败。

将 SAFE_FREE的定义加上“ {}”就可以解决上述问题了，即：

```
#define SAFE_FREE(p) { free(p); p = NULL;}
```

这样，代码

```
if(NULL != p)
    SAFE_FREE(p)
else
    ...         //do something
```

会被展开为：

```
if(NULL != p)
    { free(p); p = NULL; }
else
    ...         //do something
```

由于在 C程序中，每个语句后面加分号是一种约定俗成的习惯，因此，如下代码：

```
if(NULL != p)
    SAFE_FREE(p);
else
    ...//do something
```

将被扩展为：

```
if(NULL != p)
    { free(p); p = NULL; };
else
    ...//do something
```

这样， else分支就又没有对应的 if了，编译将无法通过。假设用了do{}while（ 0），情况就不一样了，同样的代码会被展开为：

```
if(NULL != p)
    do{ free(p); p = NULL;} while(0);
else
    ...//do something
```

不会再出现编译问题。这里读者应可以清楚地认识到， do{}while（ 0）的使用完全是为了保证宏定义的使用者能无编译错误地使用宏，它不对其使用者做任何假设。

最后，关于 Linux内核 C编程特性，要提醒各位读者特别注意一个地方： goto语句。 goto语句会带来编程的灵活性，但也会导致代码的混乱。在 Linux内核里，还是大量地使用了 goto语句，但它一般用在处理错误的分支上。例如：

```
if(get_gpioa()!=0)
{
    goto err0;
}
if(get_gpiob()!=0)
{
    goto err1;
}
if(get_gpioc()!=0)
{
    goto err2;
}
if(get_gpiod()!=0)
{
    goto err3;
}
```

```
...
err3:
    free_gpioc();
err2:
    free_gpiob();
err1:
    free_gpioa();
err0:
return -1;
```

4.4 Linux内核模块基础与骨架

前面多次提到“模块”的概念。模块具有以下两大特点：

1）模块本身不被编译入内核映像，从而控制了内核的大小。

2）模块一旦被加载，它就与内核中的其他部分完全一样。

为了让读者对 Linux模块有一个感性认识，先给出 "Hello World"的例子。

例 4.1:Hello World Module

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4 static int hello_init(void)
5 {
6       printk(KERN_ALERT " Hello World enter\n");
7       return 0;
8 }
9 static void hello_exit(void)
10 {
11       printk(KERN_ALERT " Hello World exit\n ");
12 }
13 module_init(hello_init);
14 module_exit(hello_exit);
15
16 MODULE_AUTHOR("Tony Liu");
17 MODULE_DESCRIPTION("A simple Hello World Module");
18 MODULE_ALIAS("a module skeleton");
```

注：为了便于分析模块代码骨架，我们在上面的代码中加上了行号。在必要的时候，后面也会这样处理。

编译上面的代码，在对于 Linux 2.6来讲，它将生成 hello.o的内核模块文件。通过下面的命令可以将该模块加载进 Linux内核：

```
#insmod./hello.o
```

在加载该模块时，将会看到以下输出信息：

```
Hello World enter
#
```

而通过以下命令，则可以将该模块从 Linux内核卸载：

```
#rmmod hello
```

在卸载该模块时，将会看到以下输出信息：

```
Hello World exit
#
```

上面的输出信息是分别由例 4.1代码的第 6行、第 11行输出的。在Linux内核中， printk的作用与上层应用开发的 printf是相似的。在开发调试中，常用这个函数来定位问题。该语句中用到的宏KERN_ALERT，是用来设置输出信息的优先级的。 Linux可以设置允许从 console输出信息的优先级。

通过 lsmod命令，可以查看 Linux内核中加载的模块，以及模块间的依赖关系，如图 4-2所示。

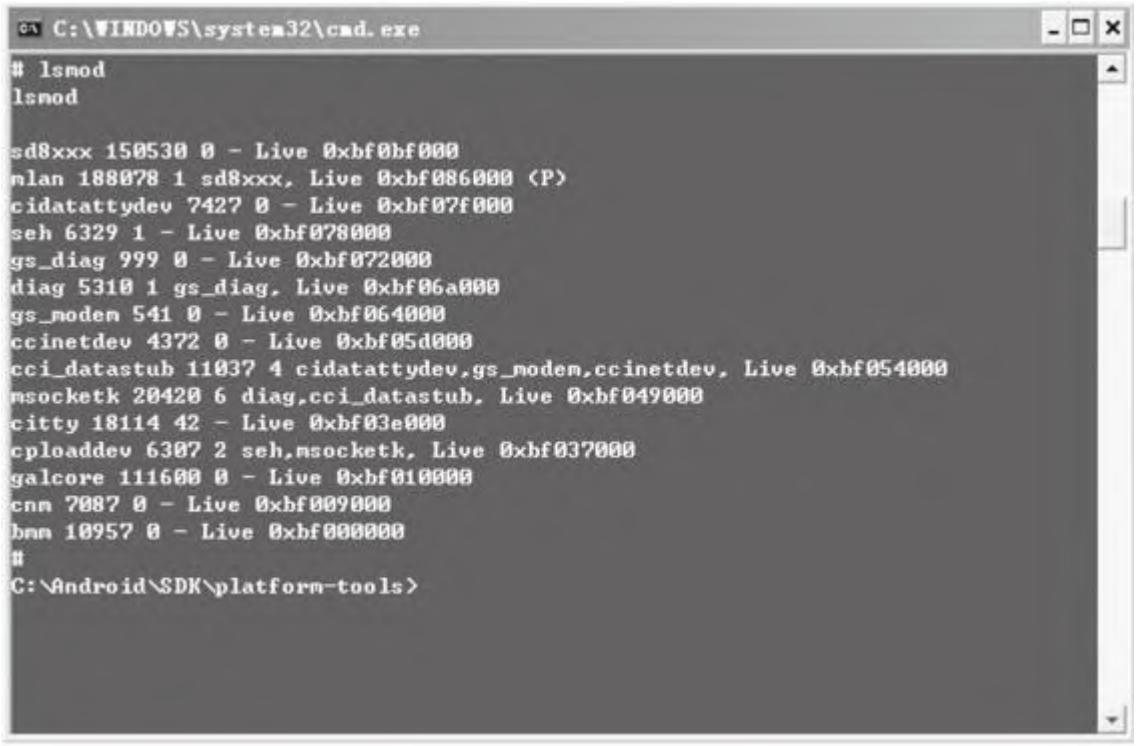


图4-2　lsmod命令

lsmod命令其实解析的是文件 /proc/modules。 hello模块加载后，会在 /sys/module目录下新生成子目录 hello。在 hello目录下又会包含 refcnt文件与 sections目录等。 refcnt记录着该模块被引用的数目，而 sections目录则记录着该模块 Image的组成。

前面讲了模块的一些基本知识，这里可以庖丁解牛，理出内核模块的骨架。 Linux内核模块主要由以下 6个部分组成：

1）模块加载函数：这部分是必需的；在模块被加载时，该函数将会被执行；一般用于完成模块的初始化工作；上面 "Hello World"例子中的第 13行就属于模块加载函数。

2）模块卸载函数：这部分也是必需的；在模块被卸载时，该函数将会被执行；其完成与模块加载相反的功能，一般用于卸载模块所占用的相关系统资源；上面 "Hello World"例子中的第 14行就属于模块卸载函数。

3）模块许可声明：这部分也是必需的；用于描述该模块的许可权限；如果不声明该许可，模块在加载时将会被警告 "kernel tainted"；

Linux内核中常用的许可是 GPL；上面 "Hello World"例子中的第 3行就属于模块许可声明。

4）模块参数：这部分是可选的；用于在模块加载时，向模块传递参数；这些参数对应着模块内的全局变量；我们将在 4.6节中给出例子。

5）模块导出符号：这部分是可选的；用于导出可供内核其他模块使用，对应于该内核里变量或函数的符号；我们将在 4.6节中给出例子。

6）模块作者等信息说明：这部分是可选的；上面 "Hello World"例子中的第 16、 17、 18行就属于模块作者、描述和别名。

4.5　Linux模块的加载与卸载

4.4节我们讲过，通过执行 insmod命令，可以加载 Linux模块。另外，通过将模块编译进 Linux内核，在内核启动时，该模块会被自动加载。事实上，模块编译时有两种模式：一种是 *模式，此种模式下，模块将直接编译进 Linux内核，成为其固有的一部分；另一种是M模式，此种模式下，模块则仅编译成一个模块，其加载要通过后期的命令或脚本来加载。 Linux内核到 2.6版本后，程序可调用 request_module（）函数加载模块，有两种方式：

```
request_module(module_name);
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

当然，调用 request_module（）函数的程序应该是一个内核模块程序。

4.4节我们还讲过，模块的加载是通过 module_init（）函数来完成的，该函数将会调用模块相关的初始化函数，如 4.4节例子中的 hello_init（）。 Linux内核初始化函数，一般会加上 __init标识声明，也就是讲， "Hello World"例子中的第 4行，最好修改为：

```
static int _ _init hello_init(void)
```

因为初始化函数在完成初始化工作之后就完成了其使命，通过 __init标识符， Linux内核将收回该函数所占的空间。初始化函数执行成功应该返回整型值 0；否则应该返回错误编码。 Linux内核默认定义的错误码都是负整型值，它们定义在 <linux/errno.h>中。

与模块加载相对应，模块的卸载也要有与模块具体相关的退出函数，如上面 "Hello World"例子中的 hello_exit（）。该类函数将完成与加载初始化相反的工作：

• 若模块加载函数注册了XXX，则模块卸载函数应该注销XXX。

• 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。

• 若模块加载函数申请了硬件资源（中断、DMA通道、I/O端口和I/O内存等）的占用，则模块卸载函数应释放这些硬件资源。

• 若模块加载函数开启了硬件，则模块卸载函数一般要关闭硬件。

在模块卸载函数上，一般会加上 __exit标识声明。因此 "Hello World"例子中的第 9行，最好修改为：

```
static void _ _exit hello_exit(void)
```

同样，这些函数完成具体模块的卸载工作后， Linux内核将根据 __exit标识符，收回该类函数所占的内存空间。

事实上，模块中的数据也有类似的修改符定义： __initdata和 __exitdata。它们分别用于定义只会在模块加载和模块卸载中会用到的数据，这些数据也将随着模块加载与模块卸载完成，其所占用的内存也将被 Linux内核收回。

4.6　Linux模块的参数与导出符号

我们可以用“ module_param（参数名，参数类型，参数读 /写权限）”为模块定义一个参数，例如，下列代码定义了一个整型参数和一个字符指针参数：

```
static char *my_name = "Tony";
static int num = 5000;
module_param(num, int, S_IRUGO);
module_param(my_name, charp, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为“ insmode模块名参数名 =参数值”，如果不传递，参数将使用模块内定义的默认值。

参数类型可以是 byte、 short、 ushort、 int、 uint、 long、 ulong、 charp（字符指针）、 bool或 invbool（布尔的求反类型），在模块被编译时会将 module_param中声明的类型与变量定义的类型进行比较，判断是否一致。

模块被加载后，在 /sys/module/目录下将出现以此模块名命名的目录。当“参数读 /写权限”为 0时，表示此参数对应不存在 sysfs文件系统下的文件节点，如果此模块存在“参数读 /写权限”不为 0的命令行参数，在此模块的目录下还将出现 parameters目录，包含一系列以参数名命名的文件节点，这些文件的权限值就是传入 module_param（）的“参数读 /写权限”，而文件的内容为参数的值。

从 Linux 2.6开始，模块也可以拥有数组型参数，形式为“ module_param_array（数组名，数组类型，数组长，参数读 /写权限）”。其中“数组长”用于限定数组的长度。在 Linux 2.6早期版本中，须将标识数组长的变量名赋给“数组长”；从 Linux 2.6.10版本后，则须将数组长变量的指针赋给“数组长”；当不需要保存实际输入的数组元素个数时，可以设置“数组长”为 NULL。

运行 insmod命令时，应使用逗号分隔输入的数组元素。

现在我们定义一个包含两个参数的模块（如例 4.2所示），并观察模块加载时被传递参数和不传递参数时的输出。

例 4.2：带参数的模块

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4
5 static char *my_name = "Tony";
6 static int num = 5000;
7
8 static int person_init(void)
9 {
10      printk(KERN_INFO " person name:%s\n",book_name);
11      printk(KERN_INFO " person num:%d\n",num);
12      return 0;
13 }
14 static void person_exit(void)
15 {
16      printk(KERN_ALERT " person module exit\n ");
17 }
18 module_init(person_init);
19 module_exit(person_exit);
20 module_param(num, int, S_IRUGO);
21 module_param(my_name, charp, S_IRUGO);
22
23 MODULE_AUTHOR("Yang Liu");
24 MODULE_DESCRIPTION("A simple Module for testing module
params");
25 MODULE_VERSION("V1.0");
```

模块可以使用如下宏导出符号到内核符号表：

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出的符号将可以被其他模块使用，使用前声明一下即可。EXPORT_SYMBOL_GPL（）只适用于包含 GPL许可权的模块。在 Linux 2.6版本中， "/proc/kallsyms"文件对应着内核符号表，它记录了符号以及符号所在的内存地址。下面例 4.3为符号导出的例子。

例 4.3：模块的符号导出

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4
5 int add_integar(int a,int b)
6 {
7       return a+b;
8 }
9
10 int sub_integar(int a,int b)
11 {
12      return a-b;
13 }
14
15 EXPORT_SYMBOL(add_integar);
16 EXPORT_SYMBOL(sub_integar);
```

如果上面的代码在某内核模块顺利执行，我们可以从"/proc/kallsyms"文件中找出 add_integar、 sub_integar相关信息。

4.7　Linux模块的使用计数

Linux 2.4内核中，模块自身通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT宏来管理自己被使用的次数。 Linux 2.6内核则提供了模块计数管理接口 try_module_get（ &module）和module_put（ &module），从而取代 Linux 2.4内核中的模块使用计数管理宏。模块的使用计数一般不必由模块自身管理，而且模块计数管理还考虑了 SMP与 PREEMPT机制的影响。

```
int try_module_get(struct module *module);
```

以上函数用于增加模块使用计数；若返回为 0，表示调用失败，希望使用的模块没有被加载或正在被卸载中。

```
void module_put(struct module *module);
```

以上函数用于减少模块使用计数。

try_module_get（）与 module_put（）的引入和使用与 Linux 2.6内核下的设备模型密切相关。 Linux 2.6内核为不同类型的设备定义了struct module*owner域，用来指向管理此设备的模块。当开始使用某个设备时，内核使用 try_module_get（ dev->owner）去增加管理此设备的 owner模块的使用计数；当不再使用此设备时，内核使用module_put（ dev->owner）减少对管理此设备的 owner模块的使用计数。这样，当设备在使用时，管理此设备的模块将不能被卸载。只有当设备不再被使用时，模块才允许被卸载。

在 Linux 2.6内核下，对于设备驱动开发工程师而言，很少需要亲自调用 try_module_get（）与 module_put（），因为此时开发人员所写的驱动通常为支持某具体设备的 owner模块，对此设备 owner模块的计数管理由内核更底层的代码（如总线驱动或是此类设备共用的核心模块）来实现，从而简化了设备驱动的开发。

第5章　Linux文件系统

在讲具体设备驱动模块之前，我们再来更深入地了解Linux的一个核心组件：文件系统。

不管是字符类设备还是块类设备，都遵循着“Linux一切都是文件”的设计思想。本章我们将更深入地讲述Linux文件系统与设备文件系统的知识。

本章的知识将对我们清楚地理解设备驱动程序很重要。因为，设备驱动最终通过Linux OS的文件系统调用而被访问。我们作为驱动程序的开发者，在编写驱动程序中将不可避免地要与设备文件系统打交道。

另外，Linux 2.4中设备文件系统实现为devfs文件系统，挂载在Linux VFS下。而Linux 2.6以后，设备文件系统则实现为基于sysfs的udev文件系统。

5.1　Linux文件系统概述

本节我们讲述 Linux系统的目录结构，以及 Linux设备驱动与 Linux文件系统的关系。

5.1.1　Linux文件系统的目录结构

前面 4.1节我们介绍了 Linux内核源码的目录结构；而这里讲的目录结构则是一个处于运行态 Linux系统自身所拥有的目录结构，请读者不要将两者混淆。

进入 Linux根目录（即“ /”， Linux文件系统的入口，也是处于最高一级的目录），运行 ″ls－ l″命令，可以看到一般 Linux系统包含以下目录。

1./dev

设备文件存储目录，应用程序通过对这些文件的读写和控制就可以访问实际的设备。

2./etc

系统配置文件的所在地，一些服务器的配置文件也在这里，如用户账号及密码配置文件。

3./lib

库文件存放目录。

4./mnt

这个目录一般是用于存放存储类设备的挂载目录，这些挂载目录代表了这些存储设备的挂载点，如 /mnt/sdcard等目录。有时我们可以让系统开机自动挂载某文件系统，也可以把代表该文件系统的挂载目录放置在 /mnt下。

5./proc

操作系统在运行时，进程及内核信息（比如 CPU、硬盘分区、内存信息等）存放在这里。 /proc目录为伪文件系统 proc的挂载目录，proc并不是真正的文件系统，它存在于内存之中。

6./sbin

存放可执行文件，大多是涉及系统管理的命令，是超级权限用户 root的可执行命令存放地，普通用户无权限执行这个目录下的命令。

7./tmp

有时用户运行程序的时候会产生临时文件， /tmp用来存放临时文件。

8./var

var表示的是变化的意思，这个目录的内容经常变动。

9./sys

Linux 2.6内核所支持的 sysfs文件系统被映射在此目录。 Linux设备驱动模型中的总线、驱动和设备都可以在 sysfs文件系统中找到对应的节点。当内核检测到在系统中出现了新设备后，内核会在 sysfs文件系统中为该新设备生成一项新的记录。

10./initrd

若在启动过程中使用了 initrd映像作为临时根文件系统，则在执行完其上的 /linuxrc挂接真正的根文件系统后，原来的初始 RAM文件系统被映射到 /initrd目录。

5.1.2 设备驱动与 Linux文件系统的关联

在第 3章我们已讲过设备驱动与 Linux文件系统，但我们在本节还将对它们二者，尤其是二者之间的关联，做进一步的讲解，因为 Linux系统其实是一个与文件为中心的系统。在 Linux文件系统中最大的一个特色就是实现了一个虚拟文件系统（ VFS）。虚拟文件系统下面可以挂载各种文件系统，包括对应设备驱动的设备文件系统，如图 5-1所示。

应用程序和 VFS之间的接口是系统调用，而 VFS与磁盘文件系统以及普通设备之间的接口是 file_operations结构体成员函数，这个结构体包含对文件进行打开、关闭、读写、控制的一系列成员函数。

由于字符设备的上层没有磁盘文件系统，所以字符设备的file_operations成员函数就直接由设备驱动提供了，file_operations正是字符设备驱动的核心。

而对于块存储设备而言， ext2、 fat、 jffs2等文件系统中会实现针对 VFS的 file_operations成员函数，设备驱动层将看不到file_operations的存在。磁盘文件系统和设备驱动将对磁盘上文件的访问最终转换成对磁盘上柱面和扇区的访问。

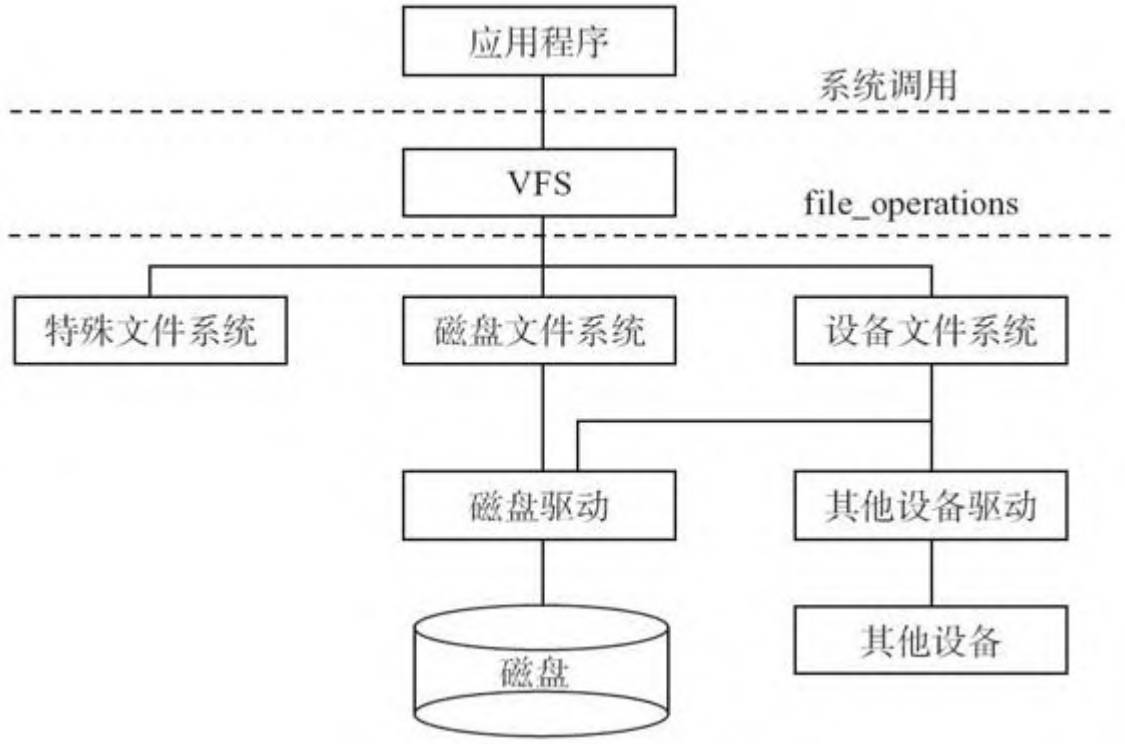


图 5-1　文件系统与设备驱动

对于我们设备驱动程序的开发来讲，须了解 file和 inode这两个结构体。

1.file结构体

文件结构体代表一个打开的文件（设备对应于设备文件），系统中每个打开的文件在内核空间都有一个关联的结构体 file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动源代码中，结构体 file的指针通常被命名为 file或 filp（即 file pointer）。代码清单 5-1给出了文件结构体的定义。

代码清单 5-1　文件结构体

```
1 struct file
2 {
3     union
4     {
5         struct list_head fu_list;
6         struct rcu_head fu_rcuhead;
7     } f_u;
8     struct dentry *f_dentry; /*与文件关联的目录入口(dentry)结构*/
9     struct vfsmount *f_vfsmnt;
10     struct file_operations *f_op; /* 和文件关联的操作*/
11     atomic_t f_count;
12     unsigned int f_flags; /*文件标志,如O_RDONLY、O_NONBLOCK、
O_SYNC*/
13     mode_t f_mode; /*文件读/写模式,FMODE_READ和FMODE_WRITE*/
14     loff_t f_pos; /* 当前读写位置*/
15     struct fown_struct f_owner;
16     unsigned int f_uid, f_gid;
17     struct file_ra_state f_ra;
18
19     unsigned long f_version;
20     void *f_security;
21
22     /* tty驱动需要,其他的驱动可能需要 */
23     void *private_data; /*文件私有数据*/
24
25 #ifdef CONFIG_EPOLL
26     /* 被fs/eventpoll.c使用以便连接所有这个文件的钩子(hooks) */
27     struct list_head f_ep_links;
28     spinlock_t f_ep_lock;
29 #endif /* #ifdef CONFIG_EPOLL */
30     struct address_space *f_mapping;
31 };
```

文件读 /写模式 f_mode、标志 f_flags都是设备驱动关心的内容，而私有数据指针 private_data在设备驱动中被广泛应用，大多被指向设备驱动自定义的，用于描述设备自身特性的结构体。

驱动程序中经常会使用如下类似的代码来检测用户打开文件的读写方式。

```
if (file->f_mode & FMODE_WRITE) //用户要求可写
{
}
if (file->f_mode & FMODE_READ) //用户要求可读
{
}
```

下面的代码可用于判断以阻塞还是非阻塞方式打开设备文件。

```
if (file->f_flags & O_NONBLOCK) //非阻塞
    pr_debug("open: non-blocking\n");
else //阻塞
    pr_debug("open: blocking\n");
```

2.inode结构体

VFS inode包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是 Linux管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁， inode结构体的定义如代码清单 5-2所示。

代码清单 5-2　inode结构体

```
1 struct inode
2 {
3     ...
4     umode_t i_mode; /* inode的权限 */
```

```
5      uid_t i_uid; /* inode拥有者的id */
6      gid_t i_gid; /* inode所属的群组id */
7      dev_t i_rdev; /* 若是设备文件,此字段将记录设备的设备号 */
8      loff_t i_size; /* inode所代表的文件大小 */
9
10      struct timespec i_atime; /* inode最近一次的存取时间 */
11      struct timespec i_mtime; /* inode最近一次的修改时间 */
12      struct timespec i_ctime; /* inode的产生时间 */
13
14      unsigned long i_blksize; /* inode在做I/O时的区块大小 */
15      unsigned long i_blocks; /* inode 所使用的block 数,一个
block 为512byte*/
16
17      struct block_device *i_bdev;
18      /*若是块设备,为其对应的block_device结构体指针*/
19      struct cdev *i_cdev; /*若是字符设备,为其对应的cdev结构体指针
*/
20      ...
21 };
```

对于表示设备文件的 inode结构， i_rdev字段包含设备编号。 Linux 2.6中设备编号分为主设备号和次设备号，前者为 dev_t的高 12位，后者为 dev_t的低 20位。下列操作用于从一个 inode中获得主设备号和次设备号：

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

查看 /proc/devices文件可以获知系统中注册的设备，第 1列为主设备号，第 2列为设备名，如图 5-2所示。

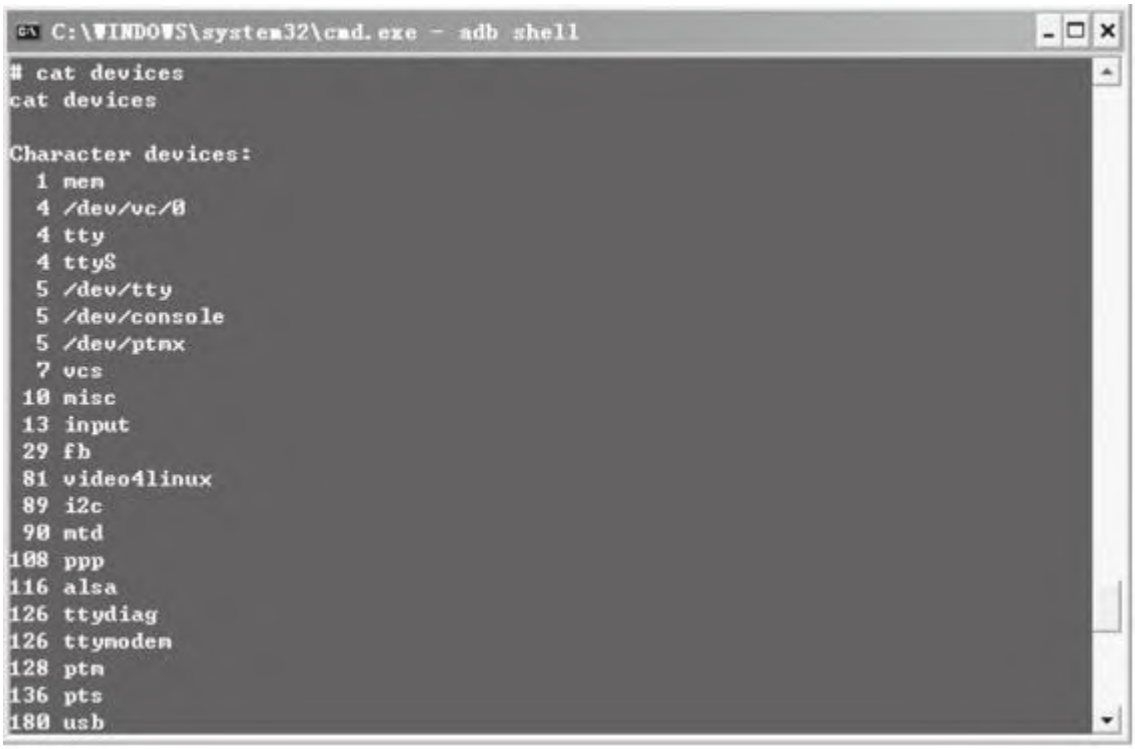


图 5-2　设备号与设备名

查看 /dev目录可以获知系统中包含的设备文件，日期的前两列给出了对应设备的主设备号和次设备号，如图 5-3所示。

从图 5-3，我们可以看到设备 ttyS0、 ttyS1、 ttyS3三个设备的主设备号都是一样的： 4。因为主设备号是与驱动对应的概念，同一类设备一般使用相同的主设备号，不同类的设备一般使用不同的主设备号。因为同一驱动可支持多个同类设备，因此用次设备号来描述使用该驱动的设备的序号，序号一般从 0开始。

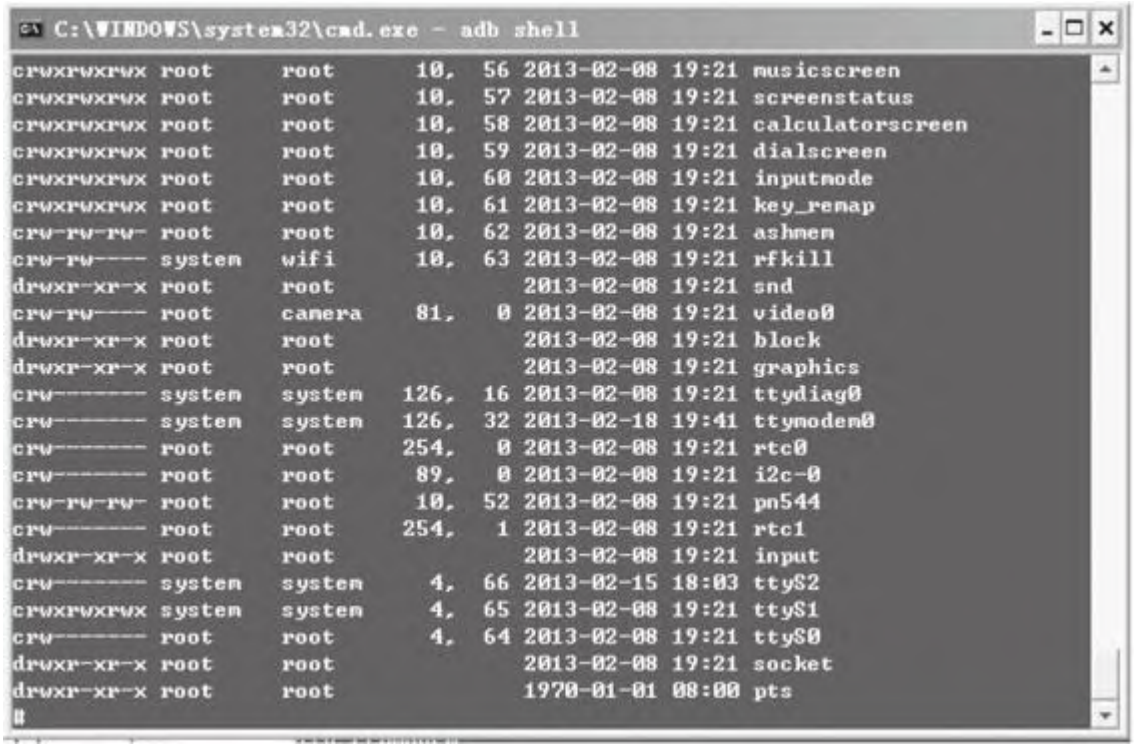


图 5-3　主设备号与次设备号

内核 documentation目录下的 devices.txt文件描述了 Linux设备号的分配情况，它由 LANANA（ The Linux Assigned Names And Numbers Authority，网址为 http://www.lanana.org/）组织维护，Torben Mathiasen是其中的主要维护者。需要注意的是， LANANA给出的设备号标准并不是硬性规定，在具体的设备驱动程序中，尽管一般会遵循 LANANA，但是也可以有例外。

5.2 Linux设备文件系统

5.2.1 devfs设备文件系统

devfs设备文件系统虽然是由 Linux 2.4内核引入的，但对我们认识 Linux 2.6的 udev设备文件系统有帮助，而在 Linux 2.6也还兼容 devfs，所以我们先讲讲 devfs。引入 devfs时，许多工程师给予了其高度评价，它的出现使得设备驱动程序能自主地管理设备文件。具体来说， devfs具有如下优点：

• 可以通过程序在设备初始化时在/dev目录下创建设备文件，卸载设备时将它删除。

• 设备驱动程序可以指定设备名、所有者和权限位，用户空间程序仍可以修改所有者和权限位。

• 不再需要为设备驱动程序分配主设备号以及处理次设备号，在程序中可以直接给register_chrdev（）传递主设备号参数的值为0，以动态获得可用的主设备号，并在devfs_register（）中指定次设备号。

驱动程序应该调用下面这些函数来进行设备文件的创建和删除工作。

```
/*创建设备目录*/
devfs_handle_t devfs_mk_dir(devfs_handle_t dir, const char
*name, void *info);
/*创建设备文件*/
devfs_handle_t devfs_register(devfs_handle_t dir, const char
*name,unsigned int flags, unsigned int major, unsigned int
minor, umode_t mode, void *ops,void *info);
/*撤销设备文件*/
void devfs_unregister(devfs_handle_t de);
```

在 Linux 2.4的设备驱动编程中，分别在模块加载和卸载函数中创建和撤销设备文件是被普遍采用并值得大力推荐的好方法。代码清单 5-3给出了一个使用 devfs的例子。

代码清单 5-3　devfs的使用范例

```
1 static devfs_handle_t devfs_handle;
2 static int _ _init xxx_init(void)
3 {
4     int ret;
5     int i;
6     /*在内核中注册设备*/
7     ret = register_chrdev(XXX_MAJOR, DEVICE_NAME, &xxx_fops);
8     if (ret < 0)
9     {
10         printk(DEVICE_NAME " can't register major
number\n");
11         return ret;
12     }
13     /*创建设备文件*/
14     devfs_handle =devfs_register(NULL, DEVICE_NAME,
DEVFS_FL_DEFAULT,
15 XXX_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &xxx_fops, NULL);
16 ...
17     printk(DEVICE_NAME " initialized\n");
18     return 0;
19 }
20
21 static void _ _exit xxx_exit(void)
22 {
23     devfs_unregister(devfs_handle); /*撤销设备文件*/
24     unregister_chrdev(XXX_MAJOR, DEVICE_NAME); /*注销设备*/
25 }
26
27 module_init(xxx_init);
28 module_exit(xxx_exit);
```

代码中第 7行和第 24行分别用于注册和注销字符设备，使用的 register_chrdev（）和 unregister_chrdev（）在 Linux 2.6内核中仍然被支持。其中， register_chrdev（）的实际意义就是将主设备号与相应设备驱动的 xxx_fops设备文件操作函数组关联上。而第 14行和第 23行分别用于创建和删除 devfs文件节点。

5.2.2　udev设备文件系统

要理解 udev设备文件系统，我们先看它与 devfs相比有何优势？

尽管 devfs有这样和那样的优点，但是，在 Linux 2.6内核中，devfs被认为是过时的方法，并最终被抛弃，由 udev取代了它。

udev完全在用户态工作，利用设备加入或移除时内核所发送的热插拔事件（ hotplug event）来工作。在热插拔时，设备的详细信息会由内核输出到位于 /sys的 sysfs文件系统。 udev的设备命名策略、权限控制和事件处理都是在用户态下完成的，它利用 sysfs中的信息来进行创建设备文件节点等工作。

由于 udev根据系统中硬件设备的状态动态更新设备文件，进行设备文件的创建和删除等，因此，在使用 udev后 /dev目录下就会只包含系统中真正存在的设备了。

devfs与 udev的另一个显著区别在于：若采用 devfs，当一个并不存在的 /dev节点被打开的时候， devfs能自动加载对应的驱动，而 udev则不能。这是因为 udev的设计者认为 Linux应该在设备被发现的时候加载驱动模块，而不是当它被访问的时候。 udev的设计者认为 devfs所提供的打开 /dev节点时自动加载驱动的功能对于一个配置正确的计算机是多余的。系统中所有的设备都应该产生热插拔事件并加载恰当的驱动，而 udev能注意到这点并且为它创建对应的设备节点。

5.2.3 sysfs文件系统与 Linux设备

1.sysfs

如上小节所述， udev设备文件系统是与 sysfs文件系统紧密相连的。

Linux 2.6内核引入了 sysfs文件系统， sysfs被看成是与 proc同类型的文件系统，即该文件系统是一个虚拟的文件系统，它可以产生一个包括所有系统硬件的分层式视图，与提供进程和状态信息的 proc文件系统十分类似。

sysfs把连接在系统上的设备和总线组织成为一个分级的文件，它们可以由用户空间存取，向用户空间导出内核数据结构以及它们的属性。sysfs的一个目的就是展示设备驱动模型中各组件的层次关系，其顶级

目录包括 block、 devices、 dev、 bus、 class、 fs、 kernel、 module、 power和 firmware。

其中， block目录包含所有的块设备， devices目录包含系统所有的设备并根据设备挂接的总线类型组织成层次结构， bus目录包含系统中所有的总线类型， class目录包含系统中的设备类型（如网卡设备、声卡设备、输入设备等）。

在 /sys/bus的 usb等子目录下，又会分出 drivers和 devices目录，该 devices目录中的文件是对 /sys/devices目录中文件的符号链接。同样的， /sys/class目录下包含许多对 /sys/devices下文件的链接。如图 5-4所示，这与设备、驱动、总线和类的现实状况是直接对应的，也正符合 Linux 2.6内核的设备模型。

随着技术的不断进步，系统的拓扑结构越来越复杂，对智能电源管理、热插拔以及即插即用的支持要求也越来越高， Linux 2.4内核已经难以满足这些需求。为适应这种形势的需要， Linux 2.6内核开发了上述全新的设备、总线、类和驱动环环相扣的设备模型。

在大多数情况下， Linux 2.6内核中的设备模型代码会处理好这些关系，内核中的总线级和其他内核子系统会完成与设备模型的交互，这使得驱动工程师几乎不需要关心设备模型。但是，理解 Linux设备模型的实现机制对驱动工程师仍然是大有裨益的，具体而言，内核将借助下文将介绍的 kobject、 kset、 subsystem、 bus_type、 device、 device_driver、 class、 class_device、 class_interface等重量级数据结构来完成设备模型的架构。

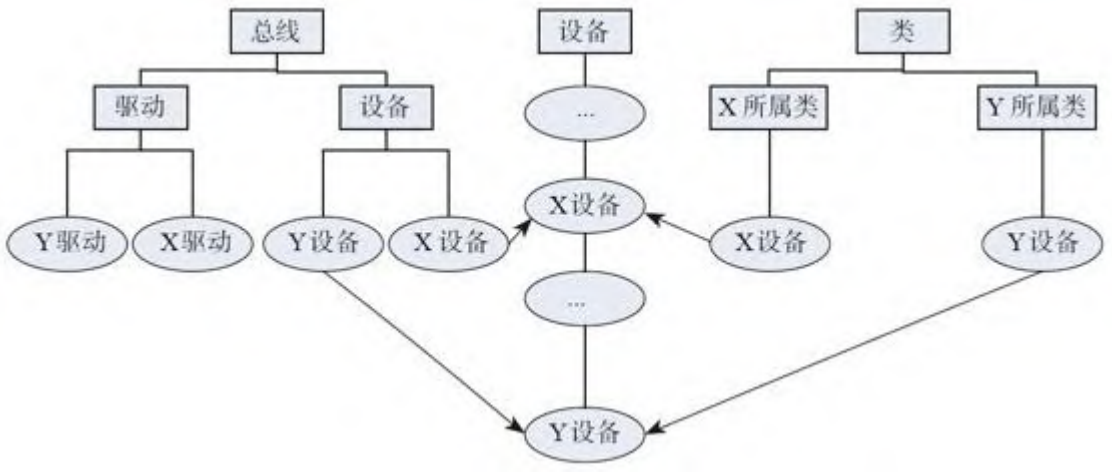


图5-4　Linux设备模型

2.kobject内核对象

kobject是 Linux 2.6引入的设备管理机制，在内核中由 kobject结构体表示，这个数据结构使所有设备在底层都具有统一的接口。kobject提供了基本的对象管理能力，是构成 Linux 2.6设备模型的核心结构，每个在内核中注册的 kobject对象都对应于 sysfs文件系统中的一个目录。 kobject结构体的定义如代码清单 5-4所示。

代码清单 5-4　kobject结构体

```
struct kobject
{
    char *k_name;
    char name[KOBJ_NAME_LEN]; //对象名称
    struct kref kref; //对象引用计数
    struct list_head entry; //用于挂接该kobject对象到kset链表
    struct kobject *parent; //指向父对象的指针
    struct kset *kset; //所属kset的指针
    struct kobj_type *ktype; //指向对象类型描述符的指针
    struct dentry *dentry; //sysfs文件系统中与该对象对应的文件节点入口
};
```

内核通过 kobject的 kref成员实现对象引用计数管理，且提供两个函数 kobject_get（）、 kobject_put（）分别用于增加和减少引用计数，当引用计数为 0时，所有该对象使用的资源将被释放。

kobject的 ktype成员是一个指向 kobj_type结构的指针，表示该对象的类型。如代码清单 5-5所示， kobj_type数据结构包含 3个成员：用于释放 kobject占用的资源的 release（）函数、指向 sysfs操作的 sysfs_ops指针和 sysfs文件系统默认属性列表。

代码清单 5-5　kobj_type结构体

```
struct kobj_type
{
    void (*release)(struct kobject *);//release函数
    struct sysfs_ops * sysfs_ops;//属性操作
    struct attribute ** default_attrs;//默认属性
};
```

---

kobj_type结构体中的 sysfs_ops包括 store（）和 show（）两个成员函数，用于实现属性的读写，代码清单 5-6给出了 sysfs_ops结构体的定义。当从用户空间读取属性时， show（）函数将被调用，该函数将指定属性值存入 buffer中返回给用户，而 store（）函数用于存储用户通过 buffer传入的属性值。与 kobject不同的是，属性在sysfs中呈现为一个文件，而 kobject则呈现为 sysfs中的目录。

代码清单 5-6　sysfs_ops结构体

```
struct sysfs_ops
{
    ssize_t (*show)(struct kobject *, struct attribute *,char
*);
    ssize_t (*store)(struct kobject *,struct attribute *,const
char *, size_t);
};
```

Linux内核中提供一系列操作 kobject的函数：

```
void kobject_init(struct kobject * kobj);
```

该函数用于初始化 kobject，它设置 kobject引用计数为 1， entry域指向自身，其所属 kset引用计数加 1。

```
int kobject_set_name(struct kobject *kobj, const char *format,
...);
```

该函数用于设置指定 kobject的名称。

```
void kobject_cleanup(struct kobject * kobj) 和void
kobject_release(struct kref *kref);
```

该函数用于清除 kobject，当其引用计数为 0时，释放对象占用的资源。

```
struct kobject *kobject_get(struct kobject *kobj);
```

该函数用于将 kobj对象的引用计数加 1，同时返回该对象的指针。

```
void kobject_put(struct kobject * kobj);
```

该函数用于将 kobj对象的引用计数减 1，如果引用计数为 0，则调用 kobject_release（）释放该 kobject对象。

```
int kobject_add(struct kobject * kobj);
```

该函数用于将 kobject对象加入 Linux设备层次，它会挂接该 kobject对象到 kset的 list链中，增加父目录各级 kobject的引用计数，在其 parent指向的目录下创建文件节点，并启动该类型内核对象的 hotplug函数。

```
int kobject_register(struct kobject * kobj);
```

该函数用于注册 kobject，它会先调用 kobject_init（）初始化 kobj，再调用 kobject_add（）完成该内核对象的添加。

```
void kobject_del(struct kobject * kobj);
```

这个函数是 kobject_add（）的反函数，它从 Linux设备层次中删除 kobject对象。

```
void kobject_unregister(struct kobject * kobj);
```

这个函数是 kobject_register（）的反函数，用于注销 kobject。与 kobject_register（）相反，它首先调用 kobject_del（）从设备层次中删除该对象，再调用 kobject_put（）减少该对象的引用计数，如果引用计数为 0，则释放该 kobject对象。

3.kset内核对象集合

kobject通常通过 kset组织成层次化的结构， kset是具有相同类型的 kobject的集合，在内核中用 kset数据结构表示，其定义如代码清单 5-7所示。

代码清单 5-7　kset结构体

```
struct kset
{
    struct subsystem * subsys; //所在的subsystem的指针
    struct kobj_type * ktype; //指向该kset对象类型描述符的指针
    struct list_head list; //用于连接该kset中所有kobject的链表头
    spinlock_t list_lock;
    struct kobject kobj; //嵌入的kobject
    struct kset_uevent_ops * uevent_ops;//事件操作集
};
```

包含在 kset中的所有 kobject被组织成一个双向循环链表， list即是该链表的头。 ktype成员指向一个 kobj_type结构，被该 kset中的所有 kobject共享，表示这些对象的类型。 kset数据结构还内嵌了一个 kobject对象（ kobj成员表示），所有属于这个 kset的 kobject对象的 parent均指向这个内嵌的对象。此外， kset还依赖于 kobj维护引用计数， kset的引用计数实际上就是内嵌的 kobject对象的引用计数。

与 kobject相似， Linux提供一系列函数操作 kset。 kset_init（）完成指定 kset的初始化， kset_get（）和 kset_put（）分别增加和减少 kset对象的引用计数， kset_add（）和 kset_del（）函数分别实现将指定 keset对象加入设备层次和从其中删除，kset_register（）函数完成 kset的注册， kset_unregister（）函数则完成 kset的注销。

kobject被创建或删除时会产生事件（ event）， kobject所属的 kset将有机会过滤事件或为用户空间添加信息。每个 kset能支持一些特定的事件变量，在热插拔事件发生时， kset的成员函数可以设置一些事件变量，这些变量将被导出到用户空间。 kset的 uevent_ops成员是执行该 kset事件操作集 kset_uevent_ops的指针，kset_uevent_ops的定义如代码清单 5-8所示。

代码清单 5-8　kset_uevent_ops结构体

```
struct kset_uevent_ops
{
    int (*filter)(struct kset *kset, struct kobject *kobj);//事件过滤
    const char *(*name)(struct kset *kset, struct kobject *kobj);
    int (*uevent)(struct kset *kset, struct kobject *kobj, char **envp,
    int num_envp, char *buffer, int buffer_size);//环境变量设置
};
```

filter（）函数用于过滤掉不需要导出到用户空间的事件，uevent（）函数用于导出一些环境变量给用户的热插拔处理程序，各

类设备导出的环境变量如下：

• USB设备：ACTION（"add"/"remove"）、DEVPATH（/sys/DEVPATH）、PRODUCT（idVendor/idProduct/bcdDevice，如46d/c281/108）、TYPE（bDeviceClass/bDeviceSubClass/bDeviceProtocol，如9/0/0）、INTERFACE（bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol，如3/1/1），如果内核配置了usbfs文件系统，还会导出DEVFS（USB驱动列表的位置，如/proc/bus/usb）和DEVICE（USB设备节点路径）。

• 网络设备：ACTION（"register"/"unregister"）、INTERFACE（接口名，如"eth0"）。

• 输入设备：ACTION（"add"/"remove"）、PRODUCT（idbus/idvendor/idproduct/idversion，如1/46d/c281/108）、NAME（设备名，如"ALCOR STRONG MAN KBD HUB"）、PHYS（设备物理地址ID，如usb-00:07.2-2.3/input0）、EV（来自evbit，如120002）、KEY（来自evbit，如e080ffdf 1dfffff ffffffff fffffe）、LED（来自ledbit，如7）。

用户空间的热插拔脚本根据传入给它的参数（如 USB的参数为热插拔程序路径、 "usb"和 0）以及内核导出的环境变量采取相应的行动，如下面的脚本程序会在 PRODUCT为 "82d/100/0"的 USB设备被插入时加载 visor模块，在被拔出时卸载 visor模块。

```
if [ "$1"="usb" ]; then
    if [ "$PRODUCT"="82d/100/0" ]; then
        if[ "$ACTION" = "add" ]; then
            /sbin/modprobe visor
        else
            /sbin/rmmod visor
        fi
    fi
fi
```

4.subsystem内核对象子系统

subsystem是一系列 kset的集合，它描述系统中某一类设备子系统，如 block_subsys表示所有的块设备，对应于 sysfs文件系统中的 block目录。 devices_subsys对应于 sysfs中的 devices目录，描述系统中所有的设备。 subsystem由结构体 subsystem数据结构描述，其定义如代码清单 5-9所示。

代码清单 5-9　subsystem结构体

```
struct subsystem
{
    struct kset kset; //内嵌的kset对象
    struct rw_semaphore rwsem; //互斥访问信号量
};
```

每个 kset必须属于某个 subsystem，通过设置 kset结构中的 subsys域指向指定的 subsystem，可以将一个 kset加入到该 subsystem。所有挂接到同一 subsystem的 kset共享同一个 rwsem信号量，用于同步访问 kset中的链表。

针对 subsystem， Linux内核也提供了一组类似于 kobject和 kset的操作函数，如下所示：

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

5.Linux设备模型组件

系统中的任一设备在设备模型中都由一个 device对象描述，其对应的数据结构 device结构体定义如代码清单 5-10所示。

代码清单 5-10　device结构体

```
struct device
{
    struct klist klist_children; // 设备列表中的孩子列表
    struct klist_node knode_parent;// 兄弟节点
    struct klist_node knode_driver; // 驱动结点
    struct klist_node knode_bus; // 总线结点
    struct device * parent; // 指向父设备
    struct kobject kobj; // 内嵌一个kobject对象
    char bus_id[BUS_ID_SIZE];// 总线上的位置
    struct device_attribute uevent_attr;

    struct semaphore sem;

    struct bus_type * bus; // 总线
    struct device_driver *driver;// 使用的驱动
    void *driver_data; // 驱动私有数据
    void *platform_data; //平台特定的数据
    void *firmware_data; //固件特定的数据(如ACPI、BIOS数据)
    struct dev_pm_info power;

    u64 *dma_mask; //DMA掩码
    u64 coherent_dma_mask;
    struct list_head dma_pools; // DMA缓冲池
    struct dma_coherent_mem *dma_mem;

    void (*release)(struct device * dev); // 释放设备方法
};
```

device结构体用于描述与设备相关的信息、设备之间的层次关系，以及设备与总线、驱动的关系。

内核提供了相应的函数用于操作 device对象。其中
device_register（）函数将一个新的 device对象插入设备模型，并自动在 /sys/devices下创建一个对应的目录。
device_unregister（）完成相反的操作，注销设备对象。
get_device（）和 put_device（）分别增加与减少设备对象的引用计数。通常 device结构体不单独使用，而是包含在更大的结构体中，比

如描述 PCI设备的 struct pci_dev，其中的 dev域就是一个 device对象。

系统中的每一个驱动程序由一个 device_driver对象描述，对应的数据结构的定义如代码清单 5-11所示。

代码清单 5-11　device_driver结构体

```
struct device_driver
{
    const char * name; //设备驱动程序的名称
    struct bus_type * bus; //总线
    struct completion unloaded;
    struct kobject kobj;//内嵌的kobject对象
    struct klist klist_devices;
    struct klist_node knode_bus;

    struct module * owner;

    int (*probe) (struct device * dev); //指向设备探测函数
    int (*remove) (struct device * dev); //指向设备移除函数
    void (*shutdown) (struct device * dev);
    int (*suspend) (struct device * dev, pm_message_t state);
    int (*resume) (struct device * dev);
};
```

与 device结构体类似， device_driver对象依靠内嵌的 kobject对象实现引用计数管理和层次结构组织。内核提供类似的函数用于操作device_driver对象。如 get_driver（）增加引用计数，driver_register（）用于向设备模型插入新的 driver对象，同时在sysfs文件系统中创建对应的目录。 device_driver结构体还包括几个函数，用于处理探测、移除和电源管理事件。

系统中总线由 bus_type结构体描述，其定义如代码清单 5-12所示。

代码清单 5-12　bus_type结构体

```
struct bus_type
{
    const char * name;//总线类型的名称
    struct subsystem subsys;//与该总线相关的subsystem
    struct kset drivers;//所有与该总线相关的驱动程序集合
    struct kset devices;//所有挂接在该总线上的设备集合
    struct klist klist_devices;
    struct klist klist_drivers;

    struct bus_attribute * bus_attrs;//总线属性
    struct device_attribute * dev_attrs;//设备属性
    struct driver_attribute * drv_attrs; //驱动程序属性

    int (*match)(struct device * dev, struct device_driver *
drv);
    int (*uevent)(struct device *dev, char **envp,
                int num_envp, char *buffer, int buffer_size);//
事件
    int (*probe)(struct device * dev);
    int (*remove)(struct device * dev);
    void (*shutdown)(struct device * dev);
    int (*suspend)(struct device * dev, pm_message_t state);
    int (*resume)(struct device * dev);
};
```

每个 bus_type对象都内嵌一个 subsystem对象， bus_subsys对象管理系统中所有总线类型的 subsystem对象。每个 bus_type对象都对应/sys/bus目录下的一个子目录，如 I2C总线类型对应于/sys/bus/i2c。在每个这样的目录下都存在两个子目录： devices和drivers（分别对应于 bus_type结构中的 devices和 drivers域）。其中 devices子目录描述连接在该总线上的所有设备，而 drivers目录则描述与该总线关联的所有驱动程序。与 device_driver对象类似， bus_type结构还包含几个处理热插拔、即插即用和电源管理事件的函数。

系统中的设备类由 class结构体描述，表示某一类设备。所有的class对象都属于 class_subsys子系统，对应于 sysfs文件系统中的/sys/class目录。代码清单 5-13给出了 class结构体的定义。

代码清单 5-13　class结构体

```
struct class
{
    const char * name; //类名
    struct module * owner;

    struct subsystem subsys; //对应的subsystem
    struct list_head children; //class_device链表
    struct list_head interfaces; //class_interface链表
    struct semaphore sem; //children和interfaces链表锁

    struct class_attribute * class_attrs;//类属性
    struct class_device_attribute * class_dev_attrs;//类设备属性

    int (*uevent)(struct class_device *dev, char **envp,
    int num_envp, char *buffer, int buffer_size);//事件

    void (*release)(struct class_device *dev);
    void (*class_release)(struct class *class);
};
```

每个 class对象包括一个 class_device链表，每个 class_device对象表示一个逻辑设备，并通过 class_device结构体中的 dev成员（一个指向 device结构体的指针）关联一个物理设备。这样，一个逻辑设备总是对应于一个物理设备，但是一个物理设备却可能对应于多个逻辑设备，代码清单 5-14给出了 class_device的定义。此外， class结构体中还包括用于处理热插拔、即插即用和电源管理事件的函数，这与 bus_type对象相似。

代码清单 5-14　class_device结构体

```
struct class_device
{
    struct list_head node;

    struct kobject kobj;//内嵌的kobject
    struct class * class; //所属的类
    dev_t devt; // dev_t
    struct class_device_attribute *devt_attr;
    struct class_device_attribute uevent_attr;
```

```
    struct device * dev; //如果存在,创建到/sys/devices 相应入口的符
号链接
    void * class_data; //私有数据
    struct class_device *parent; //父设备

    void (*release)(struct class_device *dev);
    int (*uevent)(struct class_device *dev, char **envp,
    int num_envp, char *buffer, int buffer_size);
    char class_id[BUS_ID_SIZE]; //类标识
};
```

下面两个函数用于注册和注销 class：

```
int class_register(struct class * cls);
void class_unregister(struct class * cls);
```

下面两个函数用于注册和注销 class_device：

```
int class_device_register(struct class_device *class_dev);
void class_device_unregister(struct class_device *class_dev);
```

除了 class、 class_device结构体外，还存在一个 class_interface结构体，当设备加入或退出某个对应的设备类时，将引发class_interface中的成员函数被调用， class_interface的定义如代码清单 5-15所示。

代码清单 5-15　class_interface结构体

```
struct class_interface
{
    struct list_head node;
    struct class *class;//对应的class
    int (*add)(struct class_device *, struct class_interface
*);//设备加入时触发
```

```
    void (*remove)(struct class_device *, struct
class_interface *);//设备移出时触发
};
```

下面两个函数用于注册和注销 class_interface：

```
int class_interface_register(struct class_interface
*class_intf);
void class_interface_unregister(struct class_interface
*class_intf);
```

6.属性

在 bus、 device、 driver和 class层次上都分别定义了其属性结构体，包括 bus_attribute、 driver_attribute、 class_attribute、 class_device_attribute，这几个结构体的定义在本质是完全相同的，如代码清单 5-16所示。

代码清单 5-16　bus_attribute、 driver_attribute、 class_attribute、 class_device_attribute结构体

```
/* 总线属性 */
struct bus_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct bus_type *, char * buf);
    ssize_t (*store)(struct bus_type *, const char * buf,
size_t count);
};
/* 驱动属性 */
struct driver_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct device_driver *, char * buf);
    ssize_t (*store)(struct device_driver *, const char * buf,
size_t count);
};
/* 类属性 */
```

```
struct class_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct class *, char * buf);
    ssize_t (*store)(struct class *, const char * buf, size_t
count);
};
/* 类设备属性 */
struct class_device_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct class_device *, char * buf);
    ssize_t (*store)(struct class_device *, const char * buf,
size_t count);
};
```

下面一组宏分别用于创建和初始化 bus_attribute、driver_attribute、 class_attribute、 class_device_attribute：

```
BUS_ATTR(_name,_mode,_show,_store)
DRIVER_ATTR(_name,_mode,_show,_store)
CLASS_ATTR(_name,_mode,_show,_store)
CLASS_DEVICE_ATTR(_name,_mode,_show,_store)
```

下面一组函数分别用于添加和删除 bus、 driver、 class、class_device属性：

```
int bus_create_file(struct bus_type * bus, struct bus_attribute
* attr);
void bus_remove_file(struct bus_type * bus, struct
bus_attribute *attr);
int device_create_file(struct device * dev, struct
device_attribute *attr);
void device_remove_file(struct device * dev, struct
device_attribute * attr);
int class_create_file(struct class * cls, const struct
class_attribute * attr);
void class_remove_file(struct class * cls, const struct
class_attribute * attr);
```

```
int class_device_create_file(struct class_device * class_dev,
const struct class_device_attribute * attr);
void class_device_remove_file(struct class_device * class_dev,
const struct class_device_attribute * attr);
```

---

xxx_create_file（）函数中会调用 sysfs_create_file（），而 xxx_remove_file（）函数中会调用 sysfs_remove_file（）函数，xxx_create_file（）会创建对应的 sysfs文件节点，而 xxx_remove_file（）会删除对应的 xxx文件节点。

5.2.4 udev的组成

通过网址 http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html可以详细了解 udev，从 http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/上可以下载最新的 udev包。 udev的设计目标如下：

- 在用户空间中执行。
- 动态建立/删除设备文件。
- 允许每个人都不用关心主/次设备号。
- 提供LSB标准名称。
- 如果需要，可提供固定的名称。

为了提供这些功能， udev的开发由 3个分割的子计划组成：namedev、 libsysfs和 udev。 namedev为设备命名子系统，维护设备的命名规范； libsysfs提供访问 sysfs文件系统从中获取信息的标准接口； udev提供 /dev设备节点文件的动态创建和删除策略。 udev程序承担与 namedev和 libsysfs库交互的任务，当 /sbin/hotplug程序被内核调用时， udev将被运行。 udev的工作过程如下：

1）当内核检测到在系统中出现了新设备后，内核会在 sysfs文件系统中为该新设备生成新的记录并导出一些设备特定的信息及所发生的事件。

2） udev获取内核导出的信息，它调用 namedev确定应该给该设备指定的名称，如果是新插入设备， udev将调用 libsysfs确定应该为该设备的设备文件指定的主 /次设备号，并用分析获得的设备名称和主/次设备号创建 /dev中的设备文件；如果是设备移除，则之前已经被创建的 /dev文件将被删除。

在 namedev中使用 5个步骤来确定指定设备的命名。

1）标签（ label） /序号（ serial）：这一步检查设备是否有唯一的识别记号，例如 USB设备有唯一的 USB序号， SCSI有唯一的 UUID。如果 namedev找到与这种唯一编号相对应的规则，它将使用该规则提供的名称。

2）设备总线号：这一步会检查总线设备编号，对于不可热插拔的环境，这一步足以辨别设备。例如， PCI总线编号在系统的使用期间很少变更。如果 namedev找到相对应的规则，规则中的名称就会被使用。

3）总线上的拓扑：当设备在总线上的位置匹配用户指定的规则时，就会使用该规则指定的名称。

4）替换名称：当内核提供的名称匹配指定的替代字符串时，就会使用替代字符串指定的名称。

5）内核提供的名称：如果前几个步骤都没有提供名称，默认内核被指定给该设备命名。

代码清单 5-17给出了一个 namedev命名规则的例子，第 2、 4行定义的是符合第 1步的规则，第 6、 8行定义的是符合第 2步的规则，第 11、 14行定义的是符合第 3步的规则，第 16行定义的是符合第 4步的规则。

代码清单 5-17namedev命名规则

```
1 # USB Epson printer to be called lp_epson
2 LABEL, BUS="usb", serial="HXOLL0012202323480",
NAME="lp_epson"
```

```
3 # USB HP printer to be called lp_hp,
4 LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"
5 # sound card with PCI bus id 00:0b.0 to be the first sound
card
6 NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"
7 # sound card with PCI bus id 00:07.1 to be the second sound
card
8 NUMBER, BUS="pci", id="00:07.1", NAME="dsp1"
9 # USB mouse plugged into the third port of the first hub to
be
10 # called mouse0
11 TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"
12 # USB tablet plugged into the second port of the second hub
to be
13 # called mouse1
14 TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"
15 # ttyUSB1 should always be called visor
16 REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

5.2.5 udev规则文件

udev规则文件以行为单位，以“ #”开头的行代表注释行。其余的每一行代表一个规则。每个规则分成一个或多个匹配和赋值部分。匹配部分用匹配专用的关键字来表示，相应的赋值部分用赋值专用的关键字来表示。

匹配关键字包括： ACTION（用于匹配行为）、 KERNEL（用于匹配内核设备名）、 BUS（用于匹配总线类型）、 SYSFS（用于匹配从sysfs得到的信息，比如 label、 vendor、 USB序列号）、 SUBSYSTEM（匹配子系统名）等。赋值关键字包括： NAME（创建的设备文件名）、 SYMLINK（符号创建链接名）、 OWNER（设置设备的所有者）、 GROUP（设置设备的组）、 IMPORT（调用外部程序）等。

例如如下规则：

```
SUBSYSTEM=="net", ACTION=="add",
SYSFS{address}=="00:0d:87:f6:59:f3",
IMPORT="/sbin/ rename_netiface %k eth0"
```

其中的匹配部分有 3项，分别是 SUBSYSTEM、 ACTION和 SYSFS。而赋值部分只有一项，即 IMPORT。这个规则的意思是：当系统中出现的新硬件属于 net子系统范畴，系统对该硬件采取的动作是加入这个硬件，且这个硬件在 sysfs文件系统中的 "address"信息等于 "00:0d:87:f6:59:f3"时，对这个硬件在 udev层次施行的动作是调用外部程序 /sbin/rename_netiface，并给该程序传递两个参数，一个是 "%k"，代表内核对该新设备定义的名称，另一个是 "eth0"。

通过一个简单的例子可以看出 udev和 devfs在命名方面的差异。如果系统中有两个 USB打印机，一个被称为 /dev/usb/lp0，则另外一个便是 /dev/usb/lp1。但是哪个文件对应哪个打印机是无法确定的，lp0、 lp1与实际的设备没有一一对应的关系，映射关系会因为设备发现的顺序、打印机本身关闭等原因而不确定。因此，理想的方式是两个打印机应该采用基于它们的序列号或者其他标识信息的办法来进行确定的映射， devfs无法做到这一点， udev却可以做到。使用如下规则：

```
BUS="usb", SYSFS{serial}="HXOLL0012202323480", NAME="lp_epson",
SYMLINK="printers/epson_stylus"
```

该规则中的匹配项目有 BUS和 SYSFS，赋值项目为 NAME和 SYMLINK，它意味着当一台 USB打印机的序列号为 "HXOLL0012202323480"时，创建 /dev/lp_epson文件，并同时创建一个符号链接 /dev/printers/epson_stylus。序列号为 "HXOLL0012202323480"的 USB打印机不管何时被插入，对应的设备名都是 /dev/lp_epson，而 devfs显然无法实现设备的这种固定命名。

udev规则的写法非常灵活，在匹配部分，可以通过 "*"、 "?"、 [a-c]、 [1-9]等 Shell通配符来灵活匹配多个项目。 "*"类似于 Shell中的 "*"通配符，代替任意长度的任意字符串， "?"代替一个字符，[x-y]是访问定义。此外， "%k"就是 KERNEL， "%n"则是设备的 KERNEL序号（如存储设备的分区号）。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

第6章　Linux字符设备驱动

从本章开始，基于前面的基础，我们将接触真正的Linux驱动开发。

在Linux设备驱动中，字符设备驱动较为基础。本章主要讲解Linux字符设备驱动程序的结构及其主要组成部分的编程方法。

6.1 Linux字符设备驱动结构

6.1.1 cdev结构体

在 Linux 2.6内核中使用 cdev结构体描述字符设备， cdev结构体的定义如代码清单 6-1所示。

代码清单 6-1 cdev结构体

```
struct cdev
{
    struct kobject kobj; /*内嵌的kobject对象 */
    struct module *owner; /*所属模块*/
    struct file_operations *ops; /*文件操作结构体*/
    struct list_head list;
    dev_t dev; /*设备号*/
    unsigned int count;
};
```

cdev结构体的 dev_t成员定义了设备号为 32位，其中高 12位为主设备号，低 20位为次设备号。使用下列宏可以从 dev_t中获得主设备号和次设备号。

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
```

而使用下列宏则可以通过主设备号和设备号生成 dev_t。

```
MKDEV(int major, int minor)
```

cdev结构体的另一个重要成员 file_operations定义了字符设备驱动提供给虚拟文件系统的接口函数。

同时， Linux 2.6内核提供了用于操作 cdev结构体的一组函数，如下所示：

```
void cdev_init(struct cdev *, struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

cdev_init（）函数用于初始化 cdev的成员，并建立 cdev和file_operations之间的连接，其源代码如代码清单 6-2所示。

代码清单 6-2　cdev_init（）函数

```
void cdev_init(struct cdev *cdev, struct file_operations *fops)
{
    memset(cdev, 0, sizeof *cdev);
    INIT_LIST_HEAD(&cdev->list);
    cdev->kobj.ktype = &ktype_cdev_default;
    kobject_init(&cdev->kobj);
    cdev->ops = fops; /*将传入的文件操作结构体指针赋值给cdev的ops*/
}
```

cdev_alloc（）函数用于动态申请一个 cdev内存，其源代码如代码清单 6-3所示。

代码清单 6-3　cdev_alloc（）函数

```
struct cdev *cdev_alloc(void)
{
    struct cdev *p=kmalloc(sizeof(struct cdev),GFP_KERNEL); /*
分配cdev的内存*/
```

```
    if (p) {
        memset(p, 0, sizeof(struct cdev));
        p->kobj.ktype = &ktype_cdev_dynamic;
        INIT_LIST_HEAD(&p->list);
        kobject_init(&p->kobj);
    }
    return p;
}
```

cdev_add（）函数和 cdev_del（）函数分别向系统添加和删除一个 cdev，完成字符设备的注册和注销。对 cdev_add（）的调用通常发生在字符设备驱动模块加载函数中，而对 cdev_del（）函数的调用则通常发生在字符设备驱动模块卸载函数中。

6.1.2 分配和释放设备号

在调用 cdev_add（）函数向系统注册字符设备之前，应首先调用 register_chrdev_region（）或 alloc_chrdev_region（）函数向系统申请设备号，这两个函数的原型如下：

```
int register_chrdev_region(dev_t from, unsigned count, const
char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
unsigned count, const char *name);
```

register_chrdev_region（）函数用于已知起始设备的设备号的情况，而 alloc_chrdev_region（）用于设备号未知，向系统动态申请未被占用的设备号的情况。该 alloc_chrdev_region（）函数调用成功之后，会把得到的设备号放入第一个参数 dev中。alloc_chrdev_region（）与 register_chrdev_region（）对比，其优点在于它会自动避开设备号重复的冲突。

相反，在调用 cdev_del（）函数从系统注销字符设备之后，应该调用 unregister_chrdev_region（）函数，以释放原先申请的设备号，这个函数的原型如下：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

6.1.3 file_operations结构体

file_operations结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际用来响应应用程序进行 Linux字符类设备的 open（）、 write（）、 read（）、 close（）等系统调用。file_operations结构体目前已经比较庞大，它的定义如代码清单 6-4所示。

代码清单 6-4 file_operations结构体

```
struct file_operations
{
    //拥有该结构的模块的指针,一般为THIS_MODULES
    struct module *owner;
    //用来修改文件当前的读写位置
    loff_t(*llseek)(struct file *, loff_t, int);
    //从设备中同步读取数据
    ssize_t(*read)(struct file *, char _ _user *, size_t,
loff_t*);
    //初始化一个异步的读取操作
    ssize_t(*aio_read)(struct kiocb *, char _ _user *, size_t,
loff_t);
    //向设备发送数据
    ssize_t(*write)(struct file *, const char _ _user *,
size_t, loff_t*);
    //初始化一个异步的写入操作
    ssize_t(*aio_write)(struct kiocb *, const char _ _user *,
size_t, loff_t);
    //仅用于读取目录,对于设备文件,该字段为 NULL
    int(*readdir)(struct file *, void *, filldir_t);
    //轮询函数,判断目前是否可以进行非阻塞的读取或写入
    unsigned int(*poll)(struct file *, struct
poll_table_struct*);
    //执行设备I/O控制命令
    int(*ioctl)(struct inode *, struct file *, unsigned int,
unsigned long);
    //不使用BLK文件系统,将使用此种函数指针代替ioctl
    long(*unlocked_ioctl)(struct file *, unsigned int, unsigned
```

```
long);
    //在64位系统上,32位的ioctl调用将使用此函数指针代替
    long(*compat_ioctl)(struct file *, unsigned int, unsigned
long);
    //用于请求将设备内存映射到进程地址空间
    int(*mmap)(struct file *, struct vm_area_struct*);
    //打开
    int(*open)(struct inode *, struct file*);
    int(*flush)(struct file*);
    //关闭
    int(*release)(struct inode *, struct file*);
    //刷新待处理的数据
    int(*synch)(struct file *, struct dentry *, int datasync);
    //异步fsync
    int(*aio_fsync)(struct kiocb *, int datasync);
    //通知设备FASYNC标志发生变化
    int(*fasync)(int, struct file *, int);
    int(*lock)(struct file *, int, struct file_lock*);
    // readv和writev：分散/聚集型的读写操作
    ssize_t(*readv)(struct file *, const struct iovec *,
unsigned long,loff_t*);
    ssize_t(*writev)(struct file *, const struct iovec *,
unsigned long,loff_t*);
    //下面两个函数通常为NULL,读者一般不用太关心
    ssize_t(*sendfile)(struct file *, loff_t *, size_t,
read_actor_t,void*);
    ssize_t(*sendpage)(struct file *, struct page *, int,
size_t,loff_t *, int);
    //在进程地址空间找到一个映射底层设备中的内存段的位置
    unsigned long(*get_unmapped_area)(struct file *,unsigned
long,unsigned long, unsigned long, unsigned long);
    //允许模块检查传递给fcntl(F_SETEL...)调用的标志
    int(*check_flags)(int);
    //仅对文件系统有效,驱动程序不必实现
    int(*dir_notify)(struct file *filp, unsigned long arg);
    int(*flock)(struct file *, int, struct file_lock*);
};
```

下面对 file_operations结构体中的主要成员进行进一步讲解。

llseek（）函数用来修改一个文件的当前读写位置，并将新位置返回，在出错时，这个函数返回一个负值。

read（）函数用来从设备中读取数据，成功时函数返回读取的字节数，出错时返回一个负值。

write（）函数向设备发送数据，成功时该函数返回写入的字节数。如果此函数未被实现，当用户进行 write（）系统调用时，将得到 -EINVAL返回值。

readdir（）函数仅用于目录，设备节点不需要实现它。

ioctl（）提供设备相关控制命令的实现（既不是读操作也不是写操作），当调用成功时，返回给调用程序一个非负值。内核本身识别部分控制命令，而不必调用设备驱动中的 ioctl（）。如果设备不提供 ioctl（）函数，对于内核不能识别的命令，用户进行 ioctl（）系统调用时将获得 -EINVAL返回值。

mmap（）函数将设备内存映射到进程内存中，如果设备驱动未实现此函数，用户进行 mmap（）系统调用时将获得 -ENODEV返回值。这个函数对于帧缓冲等设备特别有意义。

当用户空间调用 Linux API函数 open（）打开设备文件时，设备驱动的 open（）函数最终被调用。驱动程序可以不实现这个函数，在这种情况下，设备的打开操作永远成功。与 open（）函数对应的是 release（）函数。

poll（）函数一般用于询问设备是否可被非阻塞地立即读写。当未触发询问的条件时，用户空间进行 select（）和 poll（）系统调用将引起进程的阻塞。

aio_read（）和 aio_write（）函数分别对与文件描述符对应的设备进行异步读、写操作。设备实现这两个函数后，用户空间可以对该设备文件描述符调用 aio_read（）、 aio_write（）等系统调用进行读和写。

关于设备的阻塞访问、异步访问等概念，如果读者不是很了解，请通读第二篇。我认为对设备的访问是设备驱动的精髓。而第 7章～第 11章用较大的篇幅依序对字符设备的 I/O做详细的描述。相信读者通过这些章节的阅读，对这些概念将不再生疏。

6.1.4　Linux字符设备驱动的组成

有了 4.5节与上面知识储备后，现在可以进一步了解 Linux字符设备驱动是如何组成的。在 Linux系统中，字符设备驱动由如下几个部分组成。

1.字符设备驱动模块加载与卸载函数

在字符设备驱动模块加载函数中应完成设备号的申请和 cdev的注册，而在卸载函数中应实现设备号的释放和 cdev的注销。

有经验的驱动开发工程师通常习惯将设备定义为一个结构体，包含与设备相关的 cdev、私有数据及信号量等信息。常见的字符设备驱动模块加载和卸载函数形式如代码清单 6-5所示。

代码清单 6-5　字符设备驱动模块加载与卸载函数模板

```
//
 设备结构体

1struct xxx_dev_t
2{
3    struct cdev cdev;
4    ...
5} xxx_dev;
//设备驱动模块加载函数
6static int _ _init xxx_init(void)
7{
8     ...
9     cdev_init(&xxx_dev.cdev, &xxx_fops); //初始化cdev
10     xxx_dev.cdev.owner = THIS_MODULE;
11     //获取字符设备号
12     if (xxx_major)
13     {
14         register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
15     }
16     else
17     {
18         alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
19     }
```

```
20      ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1); //注册设备
21      ...
22}
/*设备驱动模块卸载函数*/
23static void _ _exit xxx_exit(void)
24{
25      unregister_chrdev_region(xxx_dev_no, 1); //释放占用的设备号
26      cdev_del(&xxx_dev.cdev); //注销设备
27      ...
28}
```

2.字符设备驱动的 file_operations结构体中成员函数

如前所述， file_operations结构体中成员函数是字符设备驱动与内核的接口，是用户空间对 Linux进行系统调用最终的落实者。大多数字符设备驱动会实现 read（）、 write（）和 ioctl（）函数，常见的字符设备驱动的这 3个函数如代码清单 6-6所示。

代码清单 6-6　字符设备驱动读、写、 I/O控制函数模板

```
/*读设备*/
ssize_t xxx_read(struct file *filp, char _ _user *buf, size_t
count,loff_t*f_pos)
{
    ...
    copy_to_user(buf, ..., ...);
    ...
}
/*写设备*/
ssize_t xxx_write(struct file *filp, const char _ _user *buf,
size_t count,loff_t *f_pos)
{
    ...
    copy_from_user(..., buf, ...);
    ...
}
/* ioctl函数 */
int xxx_ioctl(struct inode *inode, struct file *filp, unsigned
int cmd,unsigned long arg)
{
    ...
```

```
    switch (cmd)
    {
        case XXX_CMD1:
            ...
            break;
        case XXX_CMD2:
            ...
            break;
        default:
            /*不能支持的命令 */
            return - ENOTTY;
    }
    return 0;
}
```

设备驱动的读函数中， filp是文件结构体指针； buf是用户空间内存的地址，该地址在内核空间不能直接读写； count是要读的字节数；f_pos是读的位置相对于文件开头的偏移。

设备驱动的写函数中， filp是文件结构体指针； buf是用户空间内存的地址，该地址在内核空间不能直接读写； count是要写的字节数；f_pos是写的位置相对于文件开头的偏移。

由于内核空间与用户空间的内存不能直接互访，因此借助函数 copy_from_user（）完成用户空间到内核空间的复制，借助函数 copy_to_user（）完成内核空间到用户空间的复制。

copy_from_user（）和 copy_to_user（）函数的原型如下所示：

```
unsigned long copy_from_user(void *to, const void _ _user
*from, unsigned long count);
unsigned long copy_to_user(void _ _user *to, const void *from,
unsigned long count);
```

上述函数均返回不能被复制的字节数，因此，如果完全复制成功则返回值为 0。

如果要复制的内存是简单类型，如 char、 int、 long等，还可以使用更简单的 put_user（）和 get_user（），如下所示：

```
int val; //内核空间整型变量
...
get_user(val, (int *) arg); //用户空间到内核空间,arg是用户空间的地址
...
put_user(val, (int *) arg); //内核空间到用户空间,arg是用户空间的地址
```

读和写函数中的_ _user是一个宏，表明其后的指针指向用户空间，这个宏定义如下：

```
#ifdef _ _CHECKER_ _
# define _ _user _ _attribute_ _((noderef, address_space(1)))
#else
# define _ _user
#endif
```

I/O控制函数的 cmd参数为事先定义的 I/O控制命令，而 arg为对应于该命令的参数。例如对于串行设备，如果 SET_BAUDRATE是一个设置波特率的命令，那后面的 arg就应该是波特率值。

在字符设备驱动中，需要定义一个 file_operations的实例，并将具体设备驱动的函数赋值给 file_operations的成员，如代码清单 6-7所示。

代码清单 6-7　字符设备驱动文件操作结构体模板

```
struct file_operations xxx_fops =
{
    .owner = THIS_MODULE,
    .read = xxx_read,
    .write = xxx_write,
    .ioctl = xxx_ioctl,
    ...
};
```

上述 xxx_fops在代码清单 6-5第 9行的 cdev_init（ &xxx_dev.cdev， &xxx_fops）语句中用于建立与 cdev的连接。

图 6-1所示为字符设备驱动的结构、字符设备驱动与字符设备以及字符设备驱动与用户空间访问该设备的程序之间的关系。

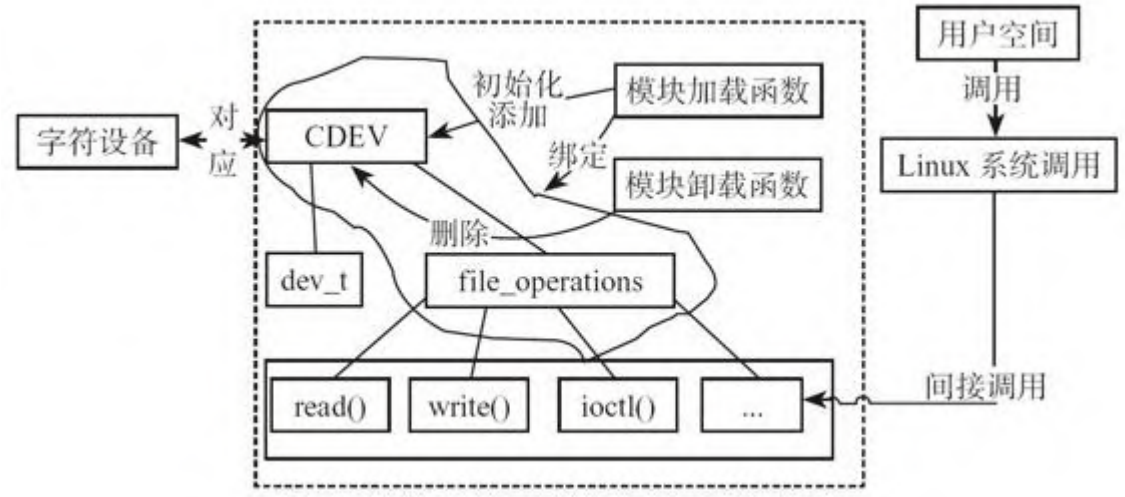


图 6-1　字符设备驱动的结构

6.2 一个字符设备驱动例子—— virtualchar

本节和下一节将基于虚拟设备 virtualchar进行字符设备驱动的讲解。在 virtualchar字符设备驱动中会分配一片大小为 MEM_SIZE（4KB）的内存空间，并在驱动中提供针对该片内存的读、写、控制和定位函数，以供用户空间的进程能通过 Linux系统调用访问这片内存。

本节将给出 virtualchar设备驱动的雏形，而后续章节会在这个雏形的基础上添加并发与同步控制等复杂功能。

6.2.1 头文件、宏及设备结构体

在 virtualchar字符设备驱动中，应包含它要使用的头文件，并定义virtualchar设备结构体及相关宏。具体参见代码清单 6-8。

代码清单 6-8 virtualchar设备结构体和宏

```
1 #include <linux/module.h>
2 #include <linux/types.h>
3 #include <linux/fs.h>
4 #include <linux/errno.h>
5 #include <linux/mm.h>
6 #include <linux/sched.h>
7 #include <linux/init.h>
8 #include <linux/cdev.h>
9 #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define MEM_SIZE 0x1000 /*全局内存大小：4KB*/
14 #define MEM_CLEAR 0x1 /*清零全局内存*/
15 #define VIRTUALCHAR_MAJOR 254 /*预设的globalmem的主设备号*/
16
17 static virtualchar_major = VIRTUALCHAR_MAJOR;
18 /*virtualchar设备结构体*/
19 struct virtualchar_dev
20 {
21         struct cdev cdev; /*cdev结构体*/
22         unsigned char mem[MEM_SIZE]; /*全局内存*/
```

```
23 };
24
25 struct virtualchar_dev dev; /*设备结构体实例*/
```

从第 19～ 23行代码可以看出，定义的 virtualchar_dev设备结构体包含了对应于 virtualchar字符设备的 cdev、使用的内存 mem[MEM_SIZE]。当然，程序中并不一定要把 mem[MEM_SIZE]和 cdev包含在一个设备结构体中，但这样定义体现了面向对象程序设计中“封装”的思想，是一种良好的编程习惯。

6.2.2 加载与卸载设备驱动

virtualchar设备驱动模块的加载和卸载函数，其实现的样式和功能与模板代码清单 6-5完全一致，如代码清单 6-9所示。

代码清单 6-9 virtualchar设备驱动模块加载与卸载函数

```
1 /*virtualchar设备驱动模块加载函数*/
2 int virtualchar_init(void)
3 {
4     int result;
5     dev_t devno = MKDEV(virtualchar_major, 0);
6
7     /* 申请字符设备驱动区域*/
8     if (virtualchar_major)
9         result = register_chrdev_region(devno, 1,
"virtualchar");
10     else
11     /* 动态获得主设备号 */
12     {
13         result = alloc_chrdev_region(&devno, 0, 1,
"virtualchar");
14         virtualchar_major = MAJOR(devno);
15     }
16     if (result < 0)
17         return result;
18
19      virtualchar_setup_cdev(); //对globalmem设备进行设置,该函数
应是驱动自我完善的函数
20     return 0;
21 }
```

```
22
23 /*virtualchar设备驱动模块卸载函数*/
24 void virtualchar_exit(void)
25 {
26     cdev_del(&dev.cdev); /*删除cdev结构*/
27     unregister_chrdev_region(MKDEV(virtualchar_major, 0),
1);/*注销设备区域*/
28 }
```

可以在第 19行调用的 virtualchar_setup_cdev（）函数中完成 cdev 结构体的初始化和添加，如代码清单 6-10所示。

代码清单 6-10　初始化并添加 cdev结构体

```
1 /*初始化并添加cdev结构体*/
2 static void virtualchar_setup_cdev()
3 {
4     int err, devno = MKDEV(virtualchar_major, 0);
5
6     cdev_init(&dev.cdev, &virtualchar_fops);
7     dev.cdev.owner = THIS_MODULE;
8     dev.cdev.ops = &virtualchar_fops;
9     err = cdev_add(&dev.cdev, devno, 1);
10     if (err)
11         printk(KERN_NOTICE "Error %d adding globalmem",
err);
12 }
```

在 cdev_init（）函数中，完成 virtualchar设备与其驱动 file_operations的关联。 file_operations结构体如代码清单 6-11所示。

代码清单 6-11　 virtualchar设备驱动文件操作结构体

```
1 static const struct file_operations virtualchar_fops =
2 {
3     .owner = THIS_MODULE,
4     .llseek = virtualchar_llseek,
```

```
5      .read = virtualchar_read,
6      .write = virtualchar_write,
7      .ioctl = virtualchar_ioctl,
8  };
```

很容易理解，该结构体链接的是一些功能函数。这些功能函数将用来响应上层用户程序的调度，完成相应设备的功能。这些函数就是我们常说的驱动功能函数，它们将牵涉到很大部分驱动的业务。

6.2.3 驱动函数实现

在本节，我们将来看看驱动函数的具体实现。

1.读 /写函数

virtualchar设备驱动的读 /写函数主要是让设备结构体的 mem[]数组与用户空间交互数据，并随着访问的字节数变更，返回用户的文件读写偏移位置。读和写函数的实现分别如代码清单 6-12和代码清单 6-13所示。

代码清单 6-12　virtualchar设备驱动的读函数

```
static ssize_t virtualchar_read(struct file *filp, char _ _user
*buf,size_t count,loff_t *ppos)
{
    unsigned long p = *ppos;
    int ret = 0;

    /*分析和获取有效的读长度*/
    if (p >= MEM_SIZE) //要读的偏移位置越界
        return count ? - ENXIO: 0;
    if (count > MEM_SIZE - p)//要读的字节数太大
        count = MEM_SIZE - p;

    /*内核空间→用户空间*/
    if (copy_to_user(buf, (void*)(dev.mem + p), count))
    {
        ret = - EFAULT;
    }
    else
```

```
    {
        *ppos += count;
        ret = count;

        printk(KERN_INFO "read %d bytes(s) from %d\n", count,
p);
    }

    return ret;
}
```

代码清单 6-13　virtualchar设备驱动的写函数

```
static ssize_t virtualchar_write(struct file *filp, const char
_ _user *buf, size_t count, loff_t *ppos)
{
    unsigned long p = *ppos;
    int ret = 0;

    /*分析和获取有效的写长度*/
    if (p >= MEM_SIZE) //要写的偏移位置越界
        return count ? - ENXIO: 0;
    if (count > MEM_SIZE - p) //要写的字节数太大
        count = MEM_SIZE - p;

    /*用户空间→内核空间*/
    if (copy_from_user(dev->mem + p, buf, count))
        ret = - EFAULT;
    else
    {
        *ppos += count;
        ret = count;

        printk(KERN_INFO "written %d bytes(s) from %d\n",
count, p);
    }
    return ret;
}
```

2.seek（）函数

seek（）函数对文件定位的起始地址可以是文件开头（ SEEK_SET，0）、当前位置（ SEEK_CUR， 1）和文件尾（ SEEK_END， 2）。virtualchar实现支持从文件开头和当前位置相对偏移。

在定位的时候，应该检查用户请求的合法性，若不合法，函数返回 -EINVAL；若合法，函数返回文件的当前位置，如代码清单 6-14所示。

代码清单 6-14　virtualchar设备驱动的 seek（）函数

```
static loff_t virtualchar_llseek(struct file *filp, loff_t
offset, int orig)
{
    loff_t ret;
    switch (orig)
    {
        case 0: /*从文件开头开始偏移*/
            if (offset < 0)
            {
                ret = -EINVAL;
                break;
            }
            if ((unsigned int)offset > MEM_SIZE) //偏移越界
            {
                ret = -EINVAL;
                break;
            }
            filp->f_pos = (unsigned int)offset;
            ret = filp->f_pos;
            break;
        case 1: /*从当前位置开始偏移*/
            if ((filp->f_pos + offset) > MEM_SIZE) //偏移越界
            {
                ret = -EINVAL;
                break;
            }
            if ((filp->f_pos + offset) < 0)
            {
                ret = -EINVAL;
                break;
            }
            filp->f_pos += offset;
            ret = filp->f_pos;
            break;
```

```
        default:
            ret = -EINVAL;
    }
    return ret;
}
```

3.ioctl（）函数

（ 1） virtualchar设备驱动的 ioctl（）函数

virtualchar设备驱动的 ioctl（）函数接受 MEM_CLEAR命令，这个命令会将全局内存的有效数据长度清零，对于设备不支持的命令，ioctl（）函数应该返回 -EINVAL，如代码清单 6-15所示。

代码清单 6-15　virtualchar设备驱动的 ioctl（）函数

```
static int virtualchar_ioctl(struct inode *inodep, struct file
*filp,unsigned int cmd, unsigned long arg)
{
    switch (cmd)
    {
        case MEM_CLEAR:
            //清除全局内存
            memset(dev->mem, 0, MEM_SIZE);
            printk(KERN_INFO "Virtualchar memory is set to
zero\n");
            break;

        default:
            return -EINVAL; //其他不支持的命令
    }
    return 0;
}
```

在上述程序中， MEM_CLEAR被宏定义为 0x01，但在实际开发中，我们并不推荐简单地对命令定义为 0x0、 0x1、 0x2等类似值，因为这样会导致不同的设备驱动拥有相同的命令号。这样，如果设备 A、 B都支持 0x0、 0x1、 0x2这样的命令，假设用户本身希望给 A发 0x1命令，可是不经意间发给了 B，这个时候 B因为支持该命令，它就会执

行该命令。因此，在 Linux内核中，推荐采用一套统一的 ioctl（）命令生成方式。

（ 2） ioctl（）命令

Linux系统建议以如表 6-1所示的方式定义 ioctl（）的命令码。

表 6-1ioctl()命令码的组成

设备类型	序列号	方向	数据长度
8 位	8 位	2 位	13/14 位

命令码的设备类型字段为一个“幻数”，可以是 0～ 0xff之间的值，内核中文档 ioctl-number.txt给出了一些推荐的和已经被使用的“幻数”。我们在定义新设备驱动“幻数”时，要避免与已有的设备“幻数”相冲突。

命令码的序列号也是 8位宽。

命令码的方向字段为 2位，该字段表示数据传送的方向，可能的值是 _IOC_NONE（无数据传输）、 _IOC_READ（读）、 _IOC_WRITE（写）和 _IOC_READ|_IOC_WRITE（双向）。数据传送的方向是从应用程序的角度来看的。

命令码的数据长度字段表示涉及的用户数据的大小，这个成员的宽度依赖于硬件体系结构，通常是 13位或者 14位。

内核还定义了 _IO（）、 _IOR（）、 _IOW（）和 _IOWR（）这 4个宏来辅助生成命令。这几个宏的作用是根据传入的 type（设备类型字段）、 nr（序列号字段）、 size（数据长度字段）和宏名隐含的方向字段移位组合生成命令码。

由于 globalmem的 MEM_CLEAR命令不涉及数据传输，可以定义如下：

```
#define VIRTUALCHARMEM_MAGIC ...
#define MEM_CLEAR _IO(VIRTUALCHARMEM_MAGIC,0)
```

（ 3）预定义命令

关于 IOCTL命令，我们要特别提醒读者注意：内核中预定义了一些I/O控制命令，如果某设备驱动中包含了与预定义命令一样的命令；这些命令将被当作预定义命令被内核处理，而不是被设备驱动处理。预定义命令有如下 4种：

• FIOCLEX：即 File IOctl CLose on EXec，对文件设置专用标志，通知内核当 exec（）系统调用发生时自动关闭打开的文件。

• FIONCLEX：即 File IOctl Not CLose on EXec，与 FIOCLEX标志相反，清除由 FIOCLEX命令设置的标志。

• FIOQSIZE：获得一个文件或者目录的大小，当用于设备文件时，返回一个 ENOTTY错误。

• FIONBIO：即 File IOctl Non-Blocking I/O，这个调用修改在filp->f_flags中的 O_NONBLOCK标志。

FIOCLEX、 FIONCLEX、 FIOQSIZE和 FIONBIO这些宏的定义如下：

```
#define FIONCLEX 0x5450
#define FIOCLEX 0x5451
#define FIOQSIZE 0x5460
#define FIONBIO 0x5421
```

细心的读者从上面的定义可以看出， FIOCLEX、 FIONCLEX、FIOQSIZE和 FIONBIO的幻数为“ T（对应 ASCII码值 0x54）”。

6.2.4 驱动设备私有数据

前面 3节给出的代码完整地实现了 virtualchar雏形，在其代码中，virtualchar设备结构体 virtualchar_dev定义了全局实例 dev（见代码清单 6-8第 25行），而 virtualchar驱动中 read（）、write（）、 ioctl（）、 llseek（）函数实现了对 dev的操作。

习惯上，成熟的 Linux驱动工程师都将文件的私有数据 private_data指向设备结构体；这样， read（）、 write（）、 ioctl（）、llseek（）等函数，就可通过 private_data访问设备结构体。

为了加深读者对驱动设备私有数据的理解，下面对各函数进行少量的修改，使读者对字符设备驱动的全貌有更清楚的认识。代码清单 6-16列出了完整的且使用文件私有数据的 virtualchar设备驱动。

代码清单 6-16　使用文件私有数据的 virtualchar设备驱动

```
1 #include <linux/module.h>
2 #include <linux/types.h>
3 #include <linux/fs.h>
4 #include <linux/errno.h>
5 #include <linux/mm.h>
6 #include <linux/sched.h>
7 #include <linux/init.h>
8 #include <linux/cdev.h>
9 #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define MEM_SIZE 0x1000 /*全局内存最大4KB*/
14 #define MEM_CLEAR 0x1 /*清零全局内存*/
15 #define MEM_MAJOR 254 /*预设的virtualchar主设备号*/
16
17 static virtualchar_major = VIRTUALCHAR_MAJOR;
18 /*virtualchar设备结构体*/
19 struct virtualchar_dev
20 {
21     struct cdev cdev; /*cdev结构体*/
22     unsigned char mem[MEM_SIZE]; /*全局内存*/
23 };
24
25 struct virtualchar_dev *virtualchar_devp; /*设备结构体指针*/
26 /*文件打开函数*/
27 int virtualchar_open(struct inode *inode, struct file *filp)
28 {
29     /*将设备结构体指针赋值给文件私有数据指针*/
30     filp->private_data = virtualchar_devp;
31     return 0;
32 }
```

```
33 /*文件释放函数*/
34 int virtualchar_release(struct inode *inode, struct file
*filp)
35 {
36     return 0;
37 }
38
39 /* ioctl设备控制函数 */
40 static int virtualchar_ioctl(struct inode *inodep, struct
file *filp,unsigned  int cmd, unsigned long arg)
41 {
42     struct virtualchar_dev *dev = filp->private_data;/*获得设
备结构体指针*/
43
44 switch (cmd)
45 {
46     case MEM_CLEAR:
47         memset(dev->mem, 0, MEM_SIZE);
48         printk(KERN_INFO "Virtualchar memory is set to
zero\n");
49         break;
50
51     default:
52         return -EINVAL;
53     }
54     return 0;
55 }
56
57 /*读函数*/
58 static ssize_t virtualchar_read(struct file *filp, char _
_user *buf, size_t size, loff_t *ppos)
59 {
60     unsigned long p = *ppos;
61     unsigned int count = size;
62     int ret = 0;
63     struct virtualchar_dev *dev = filp->private_data; /*获得
设备结构体指针*/
64
65     /*分析和获取有效的写长度*/
66     if (p >= MEM_SIZE)
67     return count ? - ENXIO: 0;
68     if (count > MEM_SIZE - p)
69     count = MEM_SIZE - p;
70
71     /*内核空间→用户空间*/
72     if (copy_to_user(buf, (void*)(dev->mem + p), count))
73     {
```

```
74          ret = - EFAULT;
75      }
76      else
77      {
78          *ppos += count;
79          ret = count;
80
81          printk(KERN_INFO "read %d bytes(s) from %d\n",
count, p);
82      }
83
84      return ret;
85 }
86
87 /*写函数*/
88 static ssize_t virtualchar_write(struct file *filp, const
char _ _user *buf,  size_t size, loff_t *ppos)
89 {
90      unsigned long p = *ppos;
91      unsigned int count = size;
92      int ret = 0;
93      struct virtualchar_dev *dev = filp->private_data; /*获得
设备结构体指针*/
94
95      /*分析和获取有效的写长度*/
96      if (p >= MEM_SIZE)
97          return count ? - ENXIO: 0;
98      if (count > MEM_SIZE - p)
99          count = MEM_SIZE - p;
100
101      /*用户空间→内核空间*/
102      if (copy_from_user(dev->mem + p, buf, count))
103          ret = - EFAULT;
104      else
105      {
106          *ppos += count;
107          ret = count;
108
109          printk(KERN_INFO "written %d bytes(s) from %d\n",
count, p);
110      }
111
112          return ret;
113 }
114
115 /* seek文件定位函数 */
116 static loff_t virtualchar_llseek(struct file *filp, loff_t
```

```
offset,int orig)
117 {
118     loff_t ret = 0;
119     switch (orig)
120     {
121         case 0: /*相对文件开始位置偏移*/
122             if (offset < 0)
123             {
124                 ret = -EINVAL;
125                 break;
126             }
127             if ((unsigned int)offset > MEM_SIZE)
128             {
129                 ret = -EINVAL;
130                 break;
131             }
132             filp->f_pos = (unsigned int)offset;
133             ret = filp->f_pos;
134             break;
135         case 1: /*相对文件当前位置偏移*/
136             if ((filp->f_pos + offset) > MEM_SIZE)
137             {
138                 ret = -EINVAL;
139                 break;
140             }
141             if ((filp->f_pos + offset) < 0)
142             {
143                 ret = -EINVAL;
144                 break;
145             }
146             filp->f_pos += offset;
147             ret = filp->f_pos;
148             break;
149         default:
150             ret = -EINVAL;
151             break;
152     }
153     return ret;
154 }
155
156 /*文件操作结构体*/
157 static const struct file_operations virtualchar_fops =
158 {
159  .owner = THIS_MODULE,
160  .llseek = virtualchar_llseek,
161  .read = virtualchar_read,
162  .write = virtualchar_write,
```

```
163  .ioctl = virtualchar_ioctl,
164  .open = virtualchar_open,
165  .release = virtualchar_release,
166 };
167
168 /*初始化并注册cdev*/
169 static void virtualchar_setup_cdev(struct virtualchar_dev
*dev, int index)
170 {
171         int err, devno = MKDEV(virtualchar_major, index);
172
173         cdev_init(&dev->cdev, &virtualchar_fops);
174         dev->cdev.owner = THIS_MODULE;
175         dev->cdev.ops = &virtualchar_fops;
176         err = cdev_add(&dev->cdev, devno, 1);
177         if (err)
178             printk(KERN_NOTICE "Error %d adding LED%d",
err, index);
179     }
180
181     /*设备驱动模块加载函数*/
182     int virtualchar_init(void)
183 {
184         int result;
185         dev_t devno = MKDEV(virtualchar_major, 0);
186
187         /* 申请设备号*/
188         if (virtualchar_major)
189             result = register_chrdev_region(devno, 1,
"Virtualchar");
190         else /* 动态申请设备号 */
191         {
192             result = alloc_chrdev_region(&devno, 0, 1,
"Virtualchar");
193             virtualchar_major = MAJOR(devno);
194         }
195         if (result < 0)
196             return result;
197
198         /* 动态申请设备结构体的内存*/
199       virtualchar_devp = kmalloc(sizeof(struct
virtualchar_dev),GFP_KERNEL);
200         if (!virtualchar_devp) /*申请失败*/
201         {
202             result = - ENOMEM;
203             goto fail_malloc;
204         }
```

```
205         memset(virtualchar_devp, 0, sizeof(struct
virtualchar_dev));
206
207         virtualchar_setup_cdev(virtualchar_devp, 0);
208         return 0;
209
210 fail_malloc: unregister_chrdev_region(devno, 1);
211         return result;
212 }
213
214 /*模块卸载函数*/
215 void virtualchar_exit(void)
216 {
217         cdev_del(&virtualchar_devp->cdev); /*注销cdev*/
218         kfree(virtualchar_devp); /*释放设备结构体内存*/
219         unregister_chrdev_region(MKDEV(virtualchar_major,
0), 1); /*释放设备号*/
220 }
221
222 MODULE_AUTHOR("Tony Liu");
223 MODULE_LICENSE("Dual BSD/GPL");
224
225 module_param(virtualchar_major, int, S_IRUGO);
226
227 module_init(virtualchar_init);
228 module_exit(virtualchar_exit);
```

除了在 virtualchar_open（）函数中通过 filp->private_data=virtualchar_devp语句（见第 30行），将设备结构体指针赋值给文件私有数据指针，并在 virtualchar_read（）、virtualchar_write（）、 virtualchar_llseek（）和 virtualchar_ioctl（）函数中，通过 struct virtualchar_dev*dev=filp->private_data语句获得设备结构体指针并使用该指针操作设备结构体外，代码清单 6-16与代码清单 6-8～代码清单 6-15的程序基本相同。

代码清单 6-16仅仅作为使用 private_data的范例，通过全局指针变量 virtualchar_devp访问特定于某 virtualchar设备的专有数据，读者只会感觉到代码条理更加清晰。如果 virtualchar不只包括一个设备，而是同时包括两个或两个以上的设备，两个同类设备的差异就可通过 private_data有效地体现出来；而这两个设备都由相同的 virtualchar内核模块所驱动。

在不对代码清单 6-16中的 virtualchar_read（）、virtualchar_write（）、 virtualchar_ioctl（）等重要函数及virtualchar_fops结构体等数据结构进行任何修改的前提下，只简单地修改 virtualchar_init（）、 virtualchar_exit（）和virtualchar_open（），就可以轻松地让 virtualchar驱动使能两个同样的设备（次设备号分别为 0和 1）工作，如代码清单 6-17所示。

代码清单 6-17　支持两个 virtualchar设备的 virtualchar驱动

```
1 /*文件打开函数,将根据不同的inode打开同类但实例不同的virtualchar设备*/
2 int virtualchar_open(struct inode *inode, struct file *filp)
3 {
4     /*将设备结构体指针赋值给文件私有数据指针*/
5     struct virtualchar_dev *dev;
6
7     dev = container_of(inode->i_cdev,struct
virtualchar_dev,cdev);
8     filp->private_data = dev;
9     return 0;
10 }
11
12 /*设备驱动模块加载函数*/
13 int virtualchar_init(void)
14 {
15     int result;
16     dev_t devno = MKDEV(virtualchar_major, 0);
17
18     /* 申请设备号*/
19     if (virtualchar_major)
20         result = register_chrdev_region(devno, 2,
"Virtualchar");
21     else /* 动态申请设备号 */
22     {
23         result = alloc_chrdev_region(&devno, 0, 2,
"Virtualchar");
24         virtualchar_major = MAJOR(devno);
25     }
26     if (result < 0)
27         return result;
28
29     /* 动态申请两个设备结构体的内存*/
30     virtualchar_devp = kmalloc(2*sizeof(struct
virtualchar_dev),GFP_KERNEL);
```

```
31      if (!virtualchar_devp) /*申请失败*/
32      {
33          result = - ENOMEM;
34          goto fail_malloc;
35      }
36      memset(virtualchar_devp, 0, 2*sizeof(struct
virtualchar_dev));
37
38      virtualchar_setup_cdev(virtualchar_devp[0], 0);
39      virtualchar_setup_cdev(&virtualchar_devp[1], 1);
40      return 0;
41
42 fail_malloc: unregister_chrdev_region(devno, 1);
43      return result;
44 }
45
46 /*模块卸载函数*/
47 void virtualchar_exit(void)
48 {
49      cdev_del(&(virtualchar_devp[0].cdev));
50      cdev_del(&(virtualchar_devp[1].cdev)); /*注销cdev*/
51      kfree(virtualchar_devp); /*释放设备结构体内存*/
52      unregister_chrdev_region(MKDEV(virtualchar_major, 0),
2); /*释放设备号*/
53 }
/* 其他代码同代码清单6-16 */
```

代码清单 6-17中第 7行调用的 container_of（）的作用是通过结构体成员的指针找到对应结构体的指针，这个技巧在 Linux内核编程中十分常用。在 container_of（ inode->i_cdev， struct virtualchar_dev， cdev）语句中，传给 container_of（）的第 1个参数是结构体成员的指针，第 2个参数为整个结构体的类型，第 3个参数为传入的第 1个参数即结构体成员的类型， container_of（）返回值为整个结构体的指针。

6.3 对 virtualchar设备的访问

编译 virtualchar的设备驱动，得到 virtualchar.o文件。运行 "insmod virtualchar.o"命令加载模块，通过 "lsmod"命令，确定 virtualchar模块已被加载。再通过 "cat/proc/devices"命令查看，发现多出了主设备号为 254的 "virtualchar"字符设备驱动，如下所示：

```
# cat /proc/devices
Character devices:
1 mem
2 pty
3 ttyp
4 /dev/vc/0
4 tty
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
21 sg
29 fb
180 usb
189 usb_device
254 virtualchar
```

接下来，通过 "mknod/dev/virtualchar c 2540"命令创建 "/dev/virtualchar"设备节点，并通过 "echo'hello world'>/dev/virtualchar"命令和 "cat/dev/virtualchar"命令分别验证设备的写和读，结果证明 "hello world"字符串被正确地写入 virtualchar字符设备：

```
# echo 'hello world' > /dev/virtualchar
# cat /dev/virtualchar
hello world
```

如果启用了 sysfs文件系统，我们还会发现多出了/sys/module/virtualchar目录，该目录下的树形结构如下所示：

```
|-- refcnt
'-- sections
    |-- .bss
    |-- .data
    |-- .gnu.linkonce.this_module
    |-- .rodata
    |-- .rodata.str1.1
    |-- .strtab
    |-- .symtab
    |-- .text
    '-- _ _versions
```

refcnt记录了 virtualchar模块的引用计数， sections下包含的多个文件则给出了 virtualchar所包含的 BSS、数据段和代码段等的地址及其他信息。

对于代码清单 6-17给出的支持两个 virtualchar设备的驱动，在加载模块后需创建两个设备节点， /dev/virtualchar0对应主设备号 virtualchar_major，次设备号 0， /dev/virtualchar1对应主设备号 virtualchar_major，次设备号 1。分别读写 /dev/virtualchar0和 /dev/virtualchar1，发现都读写了正确的对应设备。

第7章　Linux设备驱动中的内存与I/O访问

从第6章的virtualchar设备例子中，读者也许会体会到：对设备的访问其实就是对设备相应内存的访问。事实上，设备常关联着数据的输入与输出，因此对设备的访问还常牵涉到设备I/O端口的访问。本章将向读者清晰地展现Linux设备驱动核心任务：设备内存与I/O的访问。

由于Linux系统中提供了复杂的内存管理功能，所以内存的概念在Linux系统中变得相对复杂，出现了常规内存、高端内存、虚拟地址、逻辑地址、总线地址、物理地址、I/O内存、设备内存、预留内存等概念。本章也将系统地讲解内存和I/O的访问编程，带领大家走出内存和I/O的概念迷宫。

虽然在3.2.2节中，我们已对Linux的内存管理做了简述。本章我们立足Linux设备驱动，更详细地描述了Linux内存的访问机制。相信读者在读完本章后，将会对Linux内存的管理有一个更基本的认识。

7.1　CPU与内存和 I/O之间的故事

7.1.1　内存空间与 I/O空间

在 x86处理器中存在着 I/O空间的概念， I/O空间是相对于内存空间而言的，它通过特定的汇编指令 IN、 OUT来访问。端口号标识了外设的寄存器地址。 Intel语法的 IN、 OUT指令格式如下：

```
IN累加器, {端口号|DX}
OUT {端口号|DX},累加器
```

而大多数嵌入式 CPU如 ARM、 PowerPC等中并不提供 I/O空间，而仅存在内存空间。内存空间可以直接通过地址、指针来访问，程序和程序运行中使用的变量和其他数据都存在于内存空间中。也就是讲，对于 I/O空间，在这些嵌入式芯片中已被纳入内存的地址空间。如果某个地址对应着设备的 I/O空间，对该特定地址的访问，就将发生对该设备的 I/O访问。

内存地址可以直接由 C语言指针操作，例如在 ARM7处理器中执行如下代码：

```
unsigned char *p = (unsigned char *)0x0000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0x0000FF00写入 11。如果该地址被配置给处理器上 SPI0的输出端口，上面的两行程序就应该往 SPI0的 SPO信号线上输出对应 11的数据波形。

即便是在 x86处理器中，虽然提供了 I/O空间，如果由我们自己设计电路板，外设仍然可以挂接在内存空间。此时， CPU可以像访问一个内存单元那样访问外设 I/O端口，而不需要设立专门的 I/O指令。从这个意义上讲，内存空间是必需的，而 I/O空间是可选的。图 7-1给

出了内存空间和 I/O空间的对比。但为区别于一般内存的访问，我们还是习惯将基于设备内存的访问称作 I/O访问。

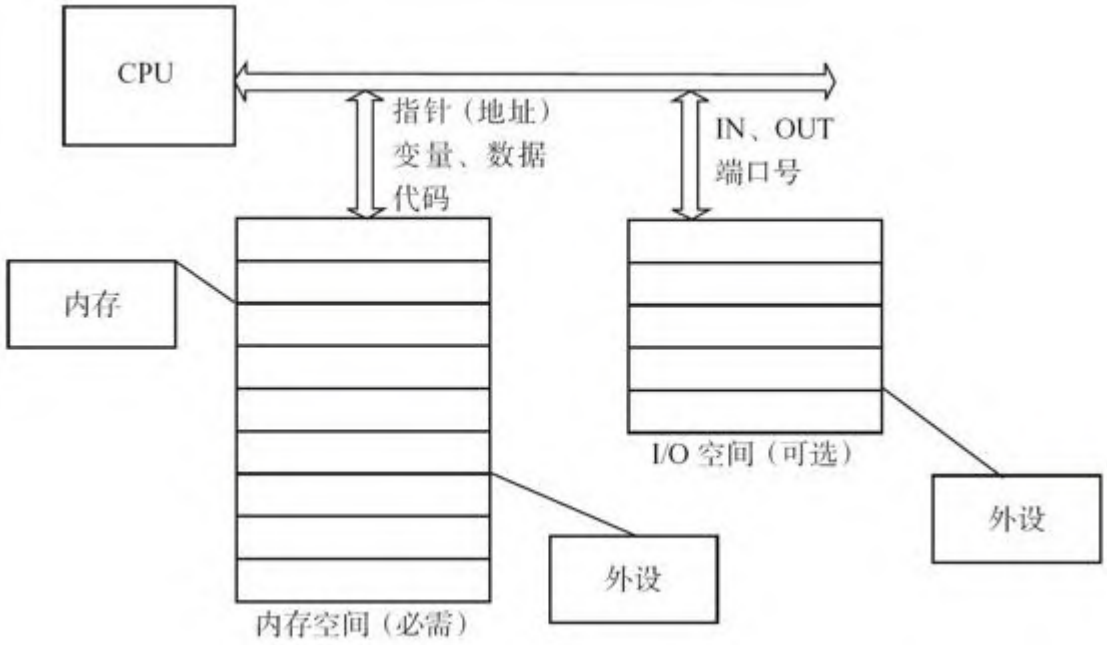


图7-1　内存空间和I/O空间

7.1.2　内存管理单元 MMU

现在高性能处理器一般都会提供一个内存管理单元（ MMU）。该硬件单元辅助操作系统进行内存管理，提供虚拟地址和物理地址的映射、内存访问权限保护和 Cache缓存控制等支持。 Linux内核借助 MMU，让用户感觉到好像程序可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中物理内存的实际容量。

为了理解基本的 MMU操作原理，须先学习几个概念。

• TLB：Translation Lookaside Buffer，即地址转换旁路缓存，TLB是MMU的核心部件，它缓存少量的虚拟地址与物理地址的转换关系，是转换表的Cache，因此也经常被称为“快表”。

• TTW：Translation Table Walk，即地址转换表漫游，完成当TLB中没有缓冲对应的地址转换关系时，通过对内存中的转换表（大多数处理器的转换表为多级页表，如图7-2所示）访问来获得虚拟地址和物理地址的对应关系的功能。TTW成功后，结果应写入TLB。

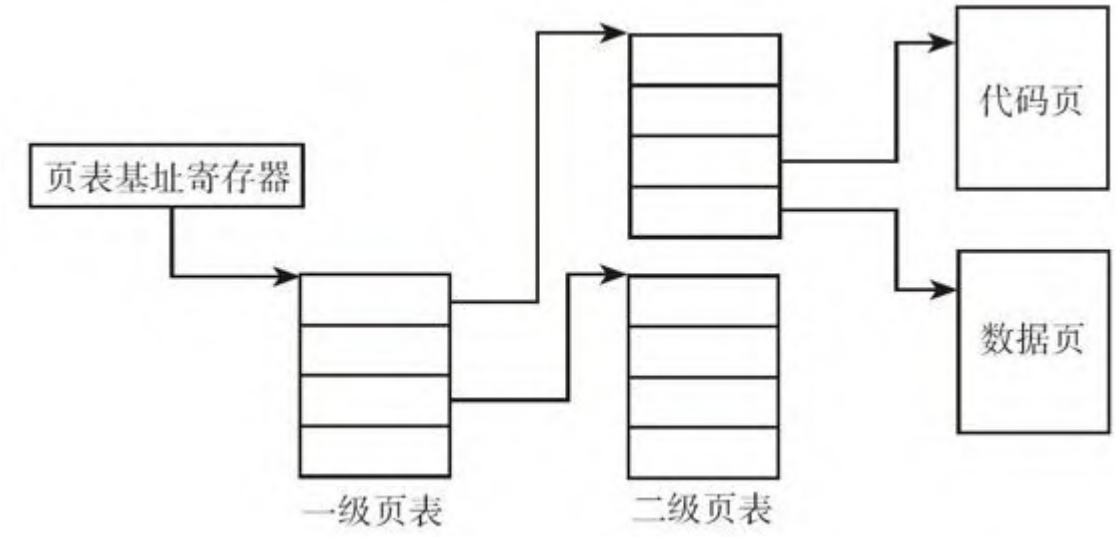


图7-2　内存中的转换表

图 7-3给出了一个典型的 ARM处理器访问内存的过程，其他处理器也执行类似过程。当 ARM要访问存储器时， MMU先查找 TLB中的虚拟地址表。如果 ARM的结构支持数据 TLB（ DTLB）和指令 TLB（ ITLB）两个独立 TLB，则除取指令使用 ITLB外，其他的都使用 DTLB。 ARM处理器的 MMU如图 7-3所示。

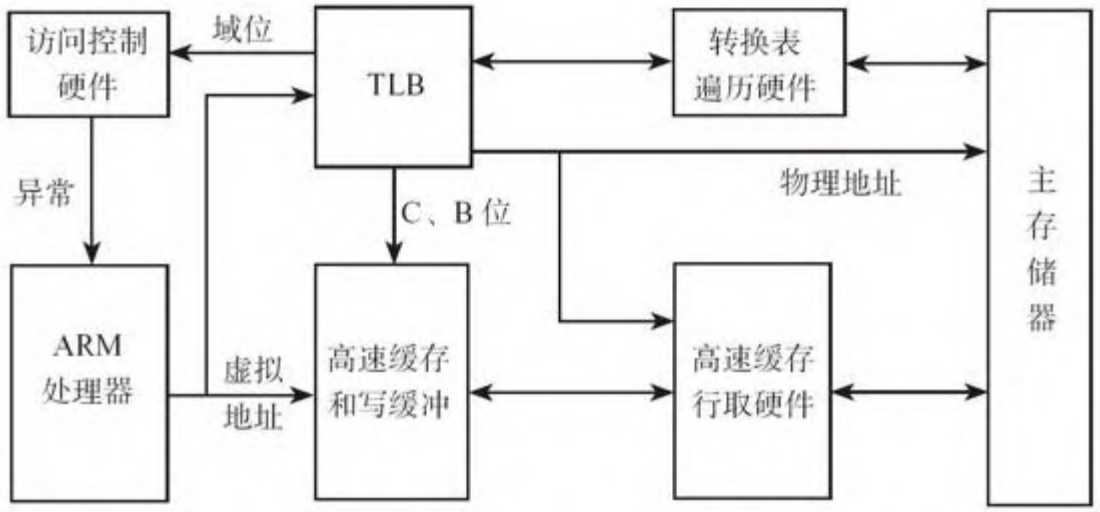


图7-3　ARM的内存管理单元

若 TLB中没有虚拟地址的入口，则转换表遍历硬件从存放于主存储器中的转换表中，获取地址转换信息和访问权限，同时将这些信息放入

TLB，它或者被放在一个未使用的入口，或者替换一个已经存在的入口。之后，在 TLB条目中控制信息的控制下，当访问权限允许时，对真实物理地址的访问将在 Cache或者在内存中发生，如图 7-4所示。

ARM中 TLB条目中的控制信息用于控制对对应地址的访问权限以及Cache的操作。

• C（高速缓存）和B（缓冲）位（见图7-3）被用来控制对应地址的高速缓存和写缓冲，并决定是否高速缓存。

• 访问权限和域位用来控制读写访问是否被允许。如果不允许，则MMU将向ARM处理器发送一个存储器异常，否则访问将被允许进行。

上面描述的 MMU机制针对的虽然是 ARM处理器，但 PowerPC、 MIPS等其他处理器也可参考。

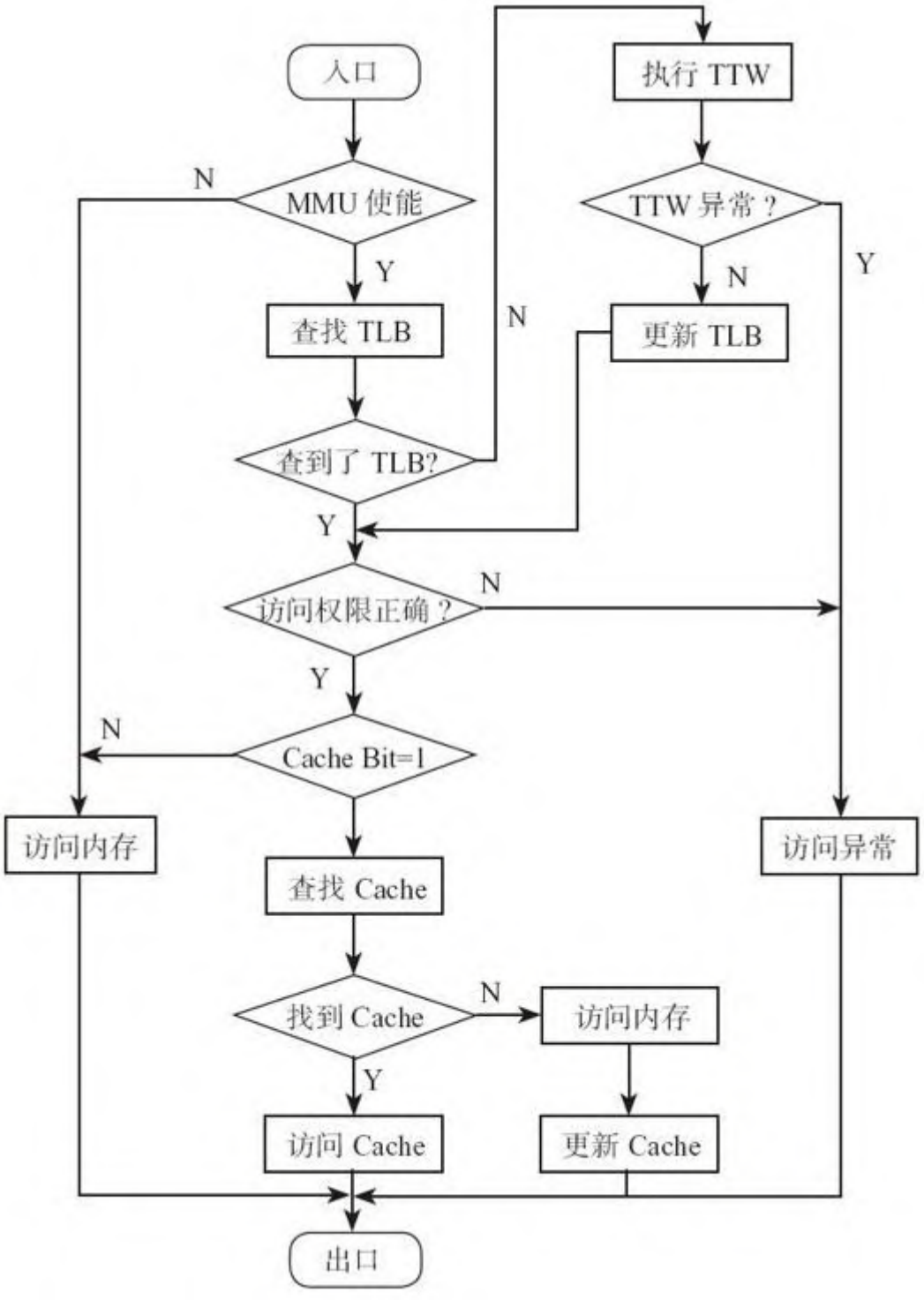


图7-4　ARM CPU进行数据访问的流程

MMU具有虚拟地址和物理地址转换、内存访问权限保护等功能。 Linux操作系统充分利用 MMU的硬件特性，为系统的每个用户进程分配独立

的内存空间，并可保证用户空间不能访问内核空间的地址，为操作系统的虚拟内存管理模块提供硬件基础。

如果处理器没有 MMU组件，例如 ARM7TDMI系列的 S3C44B0X就不带MMU，其上就无法运行旧版本的 Linux，而只能运行改版后的μCLinux。从 Linux 2.6开始，为了支持不带 MMU的处理器，融合了μCLinux，以支持 MMU-less系统，如 Dragonball、 ColdFire等。

为了加深大家对物理地址与虚拟地址映射的理解，下面我们看一个具体例子的代码。

首先，在 Linux的启动程序 Bootloader中，建立了一个 4GB虚拟地址与物理地址一一映射的一级页表，我们来追踪一下其创建过程。

Bootloader的 main（）函数首先会调用 mem_map_init（）函数以创建页表，其后调用 mmu_init（）初始化并使能 MMU。
mem_map_init（）函数如代码清单 7-1所示。

代码清单 7-1　Bootloader的 mem_map_init（）函数

```
void mem_map_init(void)
{
#ifdef CONFIG_S3C2410_NAND_BOOT
    /*CONFIG_S3C2410_NAND_BOOT = y ,在文件include/autoconf.h中定义*/
    mem_map_nand_boot();
    /*最终调用mem_mapping_linear,建立页表 */
#else
    mem_map_nor();
#endif
    cache_clean_invalidate(); /*清空Cache,使Cache无效*/
    tlb_invalidate(); /*使快表TLB无效 */
}
```

在 mem_map_nand_boot（）中会调用 mem_mapping_linear（）进行页表创建工作，如代码清单 7-2所示。

代码清单 7-2　Bootloader中的页表创建

```
static inline void mem_mapping_linear(void)
{
    unsigned long pageoffset, sectionNumber;
    putstr_hex("MMU table base address = 0x%", (unsigned
long)mmu_tlb_base);
    /* 4GB虚拟地址映射到相同的物理地址,均不能缓存 */
    for (sectionNumber = 0; sectionNumber < 4096;
sectionNumber++)
    {
        pageoffset = (sectionNumber << 20);
        *(mmu_tlb_base + (pageoffset >> 20)) = pageoffset |
MMU_SECDESC;
    }

    /*使DRAM的区域可缓存 */
/* SDRAM物理地址0x3000000-0x33ffffff,
    DRAM_BASE=0x30000000,DRAM_SIZE=64M
*/
    for (pageoffset = DRAM_BASE; pageoffset < (DRAM_BASE +
DRAM_SIZE);
        pageoffset += SZ_1M)
    {
        //DPRINTK(3, "Make DRAM section cacheable:
0x%08lx\n",pageoffset);
        *(mmu_tlb_base + (pageoffset >> 20)) = pageoffset |
MMU_SECDESC | MMU_CACHEABLE;
    }
}
```

从上面的代码可以看出是基于处理器 S3C2410的。该处理器是一个 32位的 CPU，它的地址空间为 4GB。共有 4种内存映射模式，即 Fault（无映射）、 Coarse Page（粗页表）、 Section（段）、 Fine Page（细页表）。

从代码清单 7-2来看，它使用了 S3C2410内存映射的 Section模式（实际可等同于页大小为 1MB的情况）， 4GB的虚拟空间被分成一个一个称为 Section的单位。 4GB的虚拟内存总共可以被分成 4096个段（ 1MB×4096=4GB），因此我们必须用 4096个描述符来对这组段进行描述，每个描述符占用 4个字节，故这组描述符的大小为 16KB，这 4096个描述符构成的表格就是转换表。对于 SDRAM区域，在描述符中使能了 Cache。

mmu_init（）函数直接调用 arm920_setup（）初始化 MMU，arm920_setup（）函数中则包含一系列内嵌的汇编代码，用于控制S3C2410的 CP15协处理器对 MMU的处理。

7.2　Linux内存管理

在 Linux系统中，进程的 4GB内存空间被分为两个部分——用户空间与内核空间。用户空间地址一般分布为 0～ 3GB（即 PAGE_OFFSET，在 X86中它等于 0xC0000000），剩下的 3GB～ 4GB为内核空间，如图7-5所示。通常情况下用户进程只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。用户进程只有通过系统调用（代表用户进程要进内核态执行）等方式才可以陷入内核，从而访问到内核空间。

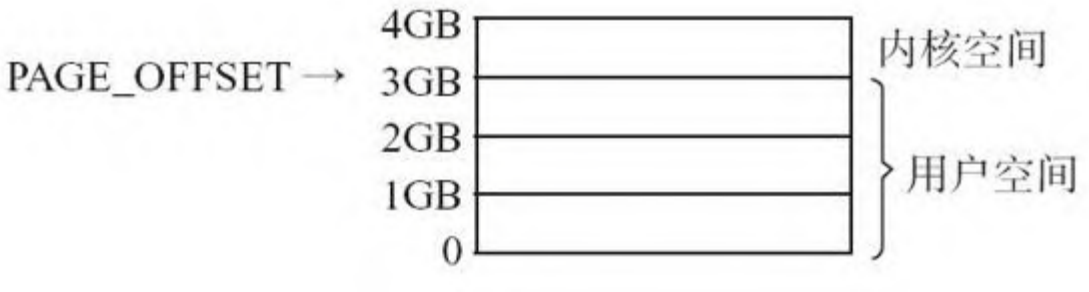


图7-5　用户空间与内核空间

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。这些页表限定且保障了各进程只能访问属于自己的内存内容。而内核空间由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，内核的虚拟空间独立于其他程序。

1GB的 Linux内核地址空间又被划分为物理内存映射区、虚拟内存分配区、高端内存映射区、专用页面映射区和系统保留映射区这几个区域，如图 7-6所示。

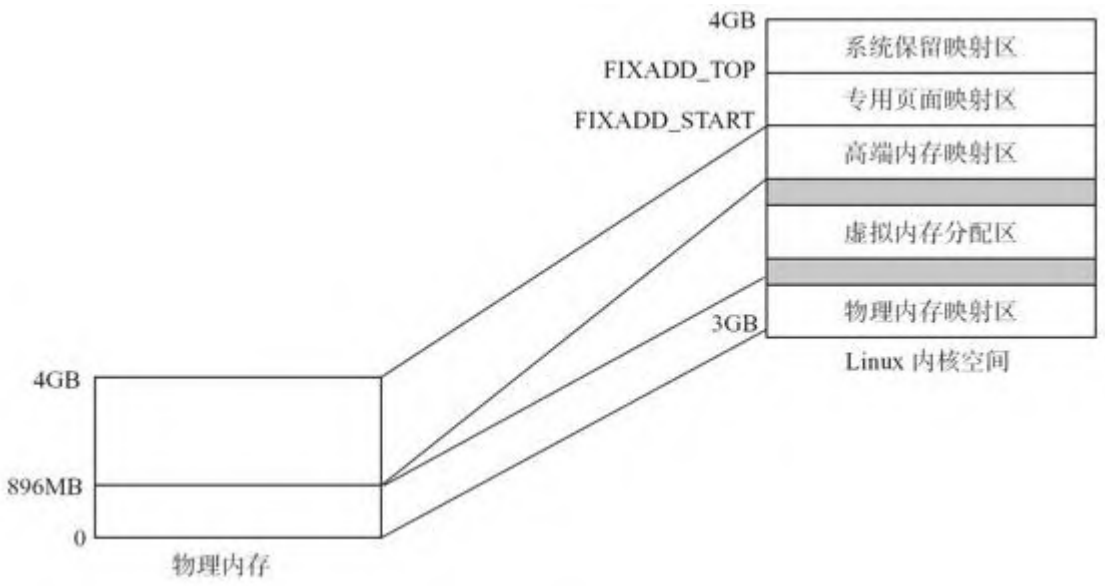


图7-6　Linux内核地址空间

一般情况下，物理内存映射区最大长度为 896MB，系统的物理内存被顺序映射在内核空间的这个区域中。当系统物理内存大于 896MB时，超过物理内存映射区的那部分内存称为高端内存（而未超过物理内存映射区的内存通常称为常规内存），内核在存取高端内存时必须将它们映射到高端内存映射区。

Linux保留内核空间最顶部 FIXADDR_TOP～ 4GB的区域作为系统保留映射区。

紧接着最顶端的保留映射区以下的一段区域为专用页面映射区（FIXADDR_START～ FIXADDR_TOP），它的总尺寸和每一页的用途由fixed_address枚举结构在编译时预定义，用 __fix_to_virt（index）可获取专用区内预定义页面的逻辑地址。其开始地址和结束地址宏定义如下：

```
#define FIXADDR_START (FIXADDR_TOP - _ _FIXADDR_SIZE)
#define FIXADDR_TOP ((unsigned long)_ _FIXADDR_TOP)
#define _ _FIXADDR_TOP 0xfffff000
```

接下来，如果系统配置了高端内存，则位于专用页面映射区之下的就是一段高端内存映射区，其起始地址为 PKMAP_BASE，定义如下：

```
#define PKMAP_BASE ( (FIXADDR_BOOT_START - PAGE_SIZE*
(LAST_PKMAP + 1))& PMD_MASK )
```

其中相关宏定义如下：

```
#define FIXADDR_BOOT_START (FIXADDR_TOP - _ _FIXADDR_BOOT_SIZE)
#define LAST_PKMAP PTRS_PER_PTE
#define PTRS_PER_PTE 512
#define PMD_MASK (~(PMD_SIZE-1))
#define PMD_SIZE (1UL << PMD_SHIFT)
#define PMD_SHIFT 21
```

在物理内存映射区和高端内存映射区之间为虚拟内存分配区（ VMALLOC_START～ VMALLOC_END），用于实现 vmalloc（）函数，它的前部与物理内存映射区有一个隔离带，后部与高端内存映射区也有一个隔离带，该区域定义如下：

```
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START (((unsigned long) high_memory +
vmalloc_earlyreserve + 2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-
1))
#ifdef CONFIG_HIGHMEM /*支持高端内存*/
# define VMALLOC_END (PKMAP_BASE-2*PAGE_SIZE)
#else /*不支持高端内存*/
# define VMALLOC_END (FIXADDR_START-2*PAGE_SIZE)
#endif
```

当系统物理内存超过 4GB时，必须使用 CPU的扩展分页（ PAE）模式所提供的 64位页目录项才能存取到 4GB以上的物理内存，这需要 CPU的支持。加入了 PAE功能的 Intel Pentium Pro及其后的 CPU允许内存最大可配置到 64GB，具备 36位物理地址空间寻址能力。

由此可见，在 3GB～ 4GB的内核空间中，从低地址到高地址依次为：物理内存映射区→隔离带→虚拟内存分配区→隔离带→高端内存映射区→专用页面映射区→系统保留映射区。

7.3 Linux内存访问

在用户空间动态申请内存的函数为 malloc（），malloc（）申请的内存由函数 free（）来释放。 malloc（）和 free（）应成对出现，即谁申请就由谁释放，否则很容易造成内存泄漏。

而在 Linux内核空间申请内存涉及的函数主要有： kmalloc（）、__get_free_pages（）和 vmalloc（）等。 kmalloc（）和__get_free_pages（）（及其类似函数）申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系。而 vmalloc（）在虚拟内存空间给出一块连续的内存区，实质上，这片连续的虚拟内存在物理内存中并不一定连续，而 vmalloc（）申请的虚拟内存和物理内存之间也不是简单的换算关系。

1.kmalloc（）

```
void *kmalloc(size_t size, int flags);
```

---

kmalloc（）的第一个参数是要分配的块的大小，第二个参数为分配标志，用于控制 kmalloc（）的行为。

最常用的分配标志是 GFP_KERNEL，其含义是在内核空间中申请内存。kmalloc（）的底层依赖 __get_free_pages（）实现，分配标志的前缀 GFP正好是这个底层函数的缩写。使用 GFP_KERNEL标志申请内存时，若暂时不能满足，则进程会睡眠等待页，即会引起阻塞，因此不能在中断上下文或持有自旋锁的时候使用 GFP_KERNEL申请内存，因为这些关键代码段被阻塞的话，极易导致系统效率降低，甚至死锁。

在中断处理函数、 tasklet和内核定时器等非进程上下文中不能阻塞，此时驱动在这些代码段应当使用 GFP_ATOMIC标志来申请内存。当使用 GFP_ATOMIC标志申请内存时，若不存在空闲页，则不等待，直接返回。

其他相对不常用的申请标志还包括 GFP_USER（用来为用户空间页分配内存，可能阻塞）、 GFP_HIGHUSER（类似 GFP_USER，但是从高端内存分配）、 GFP_NOIO（不允许任何 I/O初始化）、 GFP_NOFS（不允许进行任何文件系统调用）、 __GFP_DMA（要求分配在能够 DMA的内存区）、 __GFP_HIGHMEM（指示分配的内存可以位于高端内存）、 __GFP_COLD（请求一个较长时间不访问的页）、 __GFP_NOWARN（当一个分配无法满足时，阻止内核发出警告）、 __GFP_HIGH（高优先级请求，允许获得被内核保留给紧急状况使用的最后的内存页）、 __GFP_REPEAT（分配失败则尽力重复尝试）、 __GFP_NOFAIL（标志只许申请成功，不推荐）和 __GFP_NORETRY（若申请不到则立即放弃）。

使用 kmalloc（）申请的内存应使用 kfree（）释放，这个函数的用法与用户空间的函数 free（）类似。

2.__get_free_pages（）

__get_free_pages（）系列函数 /宏是 kmalloc（）实现的基础，__get_free_pages（）系列函数 /宏包括 get_zeroed_page（）、__get_free_page（）和 __get_free_pages（）。

```
get_zeroed_page（unsigned int flags）;
```

该函数返回一个指向新页的指针并且将该页清零。

```
__get_free_page（unsigned int flags）;
```

该宏返回一个指向新页的指针但是该页不清零，它实际上为：

```
#define __get_free_page(gfp_mask) \
_ _get_free_pages((gfp_mask),0)
```

即调用了下面的 __get_free_pages（）申请 1页。

```
_ _get_free_pages(unsigned int flags, unsigned int order);
```

该函数可分配多个页并返回分配内存的首地址，分配的页数为 2^order，分配的页也不清零。 order允许的最大值是 10（即 1024 页）或者 11（即 2048页），依赖于具体的硬件平台。

__get_free_pages（）和 get_zeroed_page（）的实现中调用了 alloc_pages（）函数， alloc_pages（）既可以在内核空间分配，也可以在用户空间分配，其原型为：

```
struct page * alloc_pages(int gfp_mask, unsigned long order);
```

参数含义与 __get_free_pages（）类似，但它返回分配的第一个页的描述符而非首地址。

使用 __get_free_pages（）系列函数 /宏申请的内存应使用下列函数释放：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

__get_free_pages等函数在使用时，其申请标志的值与 kmalloc（）完全一样，各标志的含义也与 kmalloc（）完全一致，最常用的是 GFP_KERNEL和 GFP_ATOMIC。

3.vmalloc（）

vmalloc（）一般用在为只存在于软件中（没有对应的硬件意义）的较大的顺序缓冲区分配内存， vmalloc（）远大于 __get_free_pages（）的开销，为了完成 vmalloc（），需要建立新

的页表。因此，只是调用 vmalloc（）来分配少量的内存（如 1页）是不妥的。

vmalloc（）申请的内存应使用 vfree（）释放， vmalloc（）和 vfree（）的函数原型如下：

```
void *vmalloc(unsigned long size);
void vfree(void * addr);
```

---

vmalloc（）不能用在原子上下文中，因为它的内部实现使用了标志为 GFP_KERNEL的 kmalloc（）。

使用 vmalloc（）函数的一个例子函数是 create_module（）系统调用，它利用 vmalloc（）函数来获取被创建模块需要的内存空间。

4.slab与内存池

在操作系统的运作过程中，经常会涉及大量对象的重复生成、使用和释放问题。在 Linux系统中所用到的对象，比较典型的例子是 inode、 task_struct等。如果我们能够用合适的方法使得在对象前后两次被使用时分配在同一块内存或同一类内存空间，且保留了基本的数据结构，就可以大大提高效率。 slab算法就是针对上述特点设计的。

- 创建 slab缓存。

```
struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
size_t align, unsigned long flags,
void (*ctor)(void*, struct kmem_cache *, unsigned long),
void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

---

kmem_cache_create（）用于创建一个 slab缓存，它是一个可以驻留任意数目全部同样大小的后备缓存。参数 size是要分配的每个数据结构的大小，参数 flags是控制如何进行分配的位掩码，包括

SLAB_NO_REAP（即使内存紧缺也不自动收缩这块缓存）、
SLAB_HWCACHE_ALIGN（每个数据对象对齐到一个缓存行）、
SLAB_CACHE_DMA（要求数据对象在 DMA内存区分配）等。

- 分配 slab缓存。

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

上述函数在 kmem_cache_create（）创建的 slab缓存中分配一块并返回首地址指针。

- 释放 slab缓存。

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

上述函数释放由 kmem_cache_alloc（）分配的缓存。

- 收回 slab缓存。

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

代码清单 7-3给出了 slab缓存的使用范例。

代码清单 7-3　slab缓存使用范例

```
/*创建slab缓存*/
static kmem_cache_t *xxx_cachep;
xxx_cachep = kmem_cache_create("xxx", sizeof(struct xxx), 0,
SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
/*分配slab缓存*/
struct xxx *ctx;
```

```
ctx = kmem_cache_alloc(xxx_cachep, GFP_KERNEL);
...//使用slab缓存
/*释放slab缓存*/
kmem_cache_free(xxx_cachep, ctx);
kmem_cache_destroy(xxx_cachep);
```

除了 slab缓存以外，在 Linux内核中还包含对内存池的支持，内存池技术也是一种非常经典的用于分配大量小对象的后备缓存技术。

Linux内核中，与内存池相关的操作包括如下几种。

• 创建内存池。

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t
*alloc_fn,
mempool_free_t *free_fn, void *pool_data);
```

---

mempool_create（）函数用于创建一个内存池， min_nr参数是需要预分配对象的数目， alloc_fn和 free_fn是指向内存池机制提供的标准对象分配和回收函数的指针，其原型分别为：

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

---

pool_data是分配和回收函数用到的指针， gfp_mask是分配标记。只有当指定 __GFP_WAIT标记时，分配函数才会休眠。

• 分配和回收对象。

在内存池中分配和回收对象须由以下函数来完成：

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

---

mempool_alloc（）用来分配对象，如果内存池分配器无法提供内存，那么就可以用预分配的池。

- 回收内存池。

```
void mempool_destroy（mempool_t*pool）;
```

---

mempool_create（）函数创建的内存池须由 mempool_destroy（）来回收。

7.4 Linux I/O访问

设备通常会提供控制设备、读写设备和获取设备状态的三组寄存器，即控制寄存器、数据寄存器和状态寄存器。这些寄存器可能位于 I/O空间，也可能位于内存空间。当位于 I/O空间时，通常称为 I/O端口；而位于内存空间时，对应的内存空间称为 I/O内存。

7.4.1 访问 I/O

对 I/O的访问，视 CPU的不同，就是对 I/O端口和 I/O内存的访问，实际上就是对相应设备的访问。

1.I/O端口

在 Linux设备驱动中，应使用 Linux内核提供的函数来访问定位于I/O空间的端口，这些函数包括如下几种。

- 读写字节端口（ 8位宽）。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

- 读写字端口（ 16位宽）。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

- 读写长字端口（ 32位宽）。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

• 读写一串字节。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

---

insb（）从端口 port开始读 count个字节端口，并将读取结果写入addr指向的内存； outsb（）将 addr指向的内存的 count个字节连续地写入 port开始的端口。

• 读写一串字。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

• 读写一串长字。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

上述各函数中 I/O端口 port的类型高度依赖于具体的硬件平台，因此，这些函数中参变量 port的类型使用 unsigned。

2.I/O内存

在内核中访问 I/O内存之前，需首先使用 ioremap（）函数将设备所处的物理地址映射到虚拟地址。 ioremap（）的原型如下：

```
void*ioremap（unsigned long offset，unsigned long size）；
```

---

ioremap（）与 vmalloc（）相似，也需要建立新的页表，但是它并不进行 vmalloc（）中所执行的内存分配行为。 ioremap（）返回一个特殊的虚拟地址，该地址可用来存取特定的物理地址范围。通过 ioremap（）获得的虚拟地址应该被 iounmap（）函数释放，其原型如下：

```
void iounmap（void*addr）;
```

在设备的物理地址被映射到虚拟地址之后，尽管可以直接通过指针访问这些地址，但是可以使用 Linux内核中如下一组函数来完成设备内存映射的虚拟地址的读写。

• 读 I/O内存。

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

上面的 3个函数是 Linux较早版本用来实现 I/O读操作的。虽然在 Linux 2.6中仍然可用这些函数，但我们建议开发人员尽量使用下面的新函数，来完成对设备的 I/O读操作。因为下面 3个函数具有更强的适配性与可读性。

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

• 写 I/O内存。

```
void iowrite8(u8 value, void *addr);
```

```
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

上面 3个函数则是 Linux较早版本用来实现 I/O写操作的。虽然在 Linux 2.6中仍然可用这些函数，但我们建议开发人员尽量使用下面的新函数，来完成对设备 I/O写操作。因为下面 3个函数具有更强的适配性与可读性。

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

- 读一串 I/O内存。

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
```

- 写一串 I/O内存。

```
void iowrite8_rep(void *addr, const void *buf, unsigned long
count);
void iowrite16_rep(void *addr, const void *buf, unsigned long
count);
void iowrite32_rep(void *addr, const void *buf, unsigned long
count);
```

- 复制 I/O内存。

```
void memcpy_fromio(void *dest, void *source, unsigned int
count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

• 设置 I/O内存。

```
void memset_io(void *addr, u8 value, unsigned int count);
```

3.把 I/O端口映射到内存空间

```
void *ioport_map(unsigned long port, unsigned int count);
```

通过这个函数，可以把 port开始的 count个连续的 I/O端口重映射为一段“内存空间”。然后就可以在其返回的地址上，像访问 I/O内存一样访问这些 I/O端口。当不再需要这种映射时，可以调用下面的函数来撤销：

```
void ioport_unmap(void *addr);
```

实际上，分析 ioport_map（）的源代码可发现，映射到内存空间行为实际上是给开发人员制造的一个“假象”，并没有映射到内核虚拟地址，这样做的好处是让工程师可使用统一的 I/O内存访问接口访问 I/O端口。

7.4.2　申请与释放 I/O资源

1.I/O端口申请

Linux内核提供了一组函数用于申请和释放 I/O端口。

```
struct resource*request_region（unsigned long first，unsigned long n，const char*name）；
```

这个函数向内核申请 n个端口，这些端口从 first开始， name参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL则意味着申请 I/O端口失败。

当用 request_region（）申请的 I/O端口使用完成后，应当使用 release_region（）函数将它们归还给系统，这个函数的原型如下：

```
void release_region(unsigned long start, unsigned long n);
```

2.I/O内存申请

同样， Linux内核也提供了一组函数用于申请和释放 I/O内存。

```
struct resource*request_mem_region(unsigned long start,
unsigned long len, char*name);
```

这个函数向内核申请 n个内存地址，这些地址从 start开始， name参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL则意味着申请 I/O内存失败。

当用 request_mem_region（）申请的 I/O内存使用完成后，应当使用 release_mem_region（）函数将它们归还给系统，这个函数的原型如下：

```
void release_mem_region(unsigned long start, unsigned long
len);
```

上述 request_region（）和 request_mem_region（）都不是必需的，但建议使用。其任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他驱动想再次申请该资源时就会失败。

有许多设备驱动程序在没有申请 I/O端口和 I/O内存之前就直接访问了，但这不安全。

3.I/O内存的静态映射

在将 Linux移植到目标电路板的过程中，因为设备常是固定的，如果对设备的访问通过 I/O内核的方式，通常会在系统启动时就建立好外设 I/O内存物理地址到虚拟地址的静态映射，这个映射通过在电路板对应的 map_desc结构体数组中添加新的成员来完成， map_desc结构体的定义如代码清单 7-4所示。

代码清单 7-4　map_desc结构体

```
struct map_desc
{
    unsigned long virtual;//虚拟地址
    unsigned long pfn ;// __phys_to_pfn(phy_addr)
    unsigned long length; //大小
    unsigned int type;//类型
};
```

例如，某 ARM开发板的 map_desc结构体数组定义如代码清单 7-5所示，位于文件 /arch/arm/mach-xxx.c。

代码清单 7-5　xxx开发板的 map_desc数组

```
static struct map_desc xxx_iodesc[] __initdata = {
{    /* Devs */
        .virtual    =  0xf2000000,
        .pfn        = __phys_to_pfn(0x40000000),
        .length        = 0x02000000,
        .type        = MT_DEVICE
    }, {    /* Mem Ctl */
        .virtual    =  0xf6000000,
        .pfn        = __phys_to_pfn(0x48000000),
        .length        = 0x00200000,
        .type        = MT_DEVICE
    }, {    /* Camera */
```

```
        .virtual    =  0xfa000000,
        .pfn        = __phys_to_pfn(0x50000000),
        .length        = 0x00100000,
        .type        = MT_DEVICE
    }, {    /* Sys */
        .virtual    = 0xfb000000,
        .pfn        = __phys_to_pfn(0x46000000),
        .length        = 0x00010000,
        .type        = MT_DEVICE,
    }, {    /* IMem ctl */
        .virtual    =  0xfe000000,
        .pfn        = __phys_to_pfn(0x58000000),
        .length        = 0x00100000,
        .type        = MT_DEVICE
    }, {    /* UNCACHED_PHYS_0 */
        .virtual    = 0xff000000,
        .pfn        = __phys_to_pfn(0x00000000),
        .length        = 0x00100000,
        .type        = MT_DEVICE
    }
};
```

Linux操作系统移植到特定平台上， MACHINE_START到 MACHINE_END宏之间的定义针对特定电路板而设计，其中的 map_io（）成员函数就用来完成 I/O内存的静态映射，代码清单 7-6给出了 xxx电路板的 MACHINE_START、 MACHINE_END宏的例子。

代码清单 7-6　xxx MACHINE_START、 MACHINE_END宏

```
1 MACHINE_START(TTC_DKB, "PXA910-based TTC_DKB Development
Platform")
2 .phys_io        = APB_PHYS_BASE,
3 .io_pg_offst    = (APB_VIRT_BASE >> 18) & 0xfffc,
4 .map_io        = pxa_map_io,
5
6 .nr_irqs    = TTCDKB_NR_IRQS,
7 .init_irq       = pxa910_init_irq,
8 .timer          = &pxa910_timer,
9 .init_machine   = ttc_dkb_init,
10 MACHINE_END
```

第 4行赋值给 map_io的 pxa_map_io（）函数完成开发板 I/O内存的静态映射，最终调用的是 cpu->map_io（）以建立 map_desc数组中物理内存和虚拟内存的静态映射关系。

在一个已经移植好 OS的内核中，驱动开发工程师完全可以对非常规内存区域的 I/O内存（外设控制器寄存器、 MCU内部集成的外设控制器寄存器等）依照电路板的资源使用情况添加到 map_desc数组中。

此后，在设备驱动中访问经过 map_desc数组映射后的 I/O内存时，直接在 map_desc数组中该段的虚拟地址上加上相应的偏移即可，不再需要使用 ioremap（）。

7.4.3 I/O访问流程

在申请好 I/O资源之后，我们就可以对相应设备进行 I/O访问了。

I/O端口访问的一种途径是直接使用 I/O端口操作函数：在设备打开或驱动模块被加载时申请 I/O端口区域，之后使用 inb（）、 outb（）等进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O端口范围。整个流程如图 7-7所示。

图 7-7 I/O端口的访问流程（不映射到内存空间）

I/O端口访问的另一种途径是将 I/O端口映射为内存进行访问：在设备打开或驱动模块被加载时，申请 I/O端口区域并使用 ioport_map（）

映射到内存，之后使用 I/O内存的函数进行端口访问，最后，在设备关闭或设备驱动模块被卸载时释放 I/O端口并释放映射。整个流程如图 7-8所示。

I/O内存的访问步骤：首先调用 request_mem_region（）申请资源，接着将寄存器地址通过 ioremap（）映射到内核空间虚拟地址，之后就可以通过 Linux设备访问编程接口访问这些设备的寄存器了。访问完成后，应对 ioremap（）申请的虚拟地址进行释放，并释放 release_mem_region（）申请的 I/O内存资源。

7.4.4　设备地址与用户空间的映射

本节的内容很重要，敬请读者耐心地读完它。

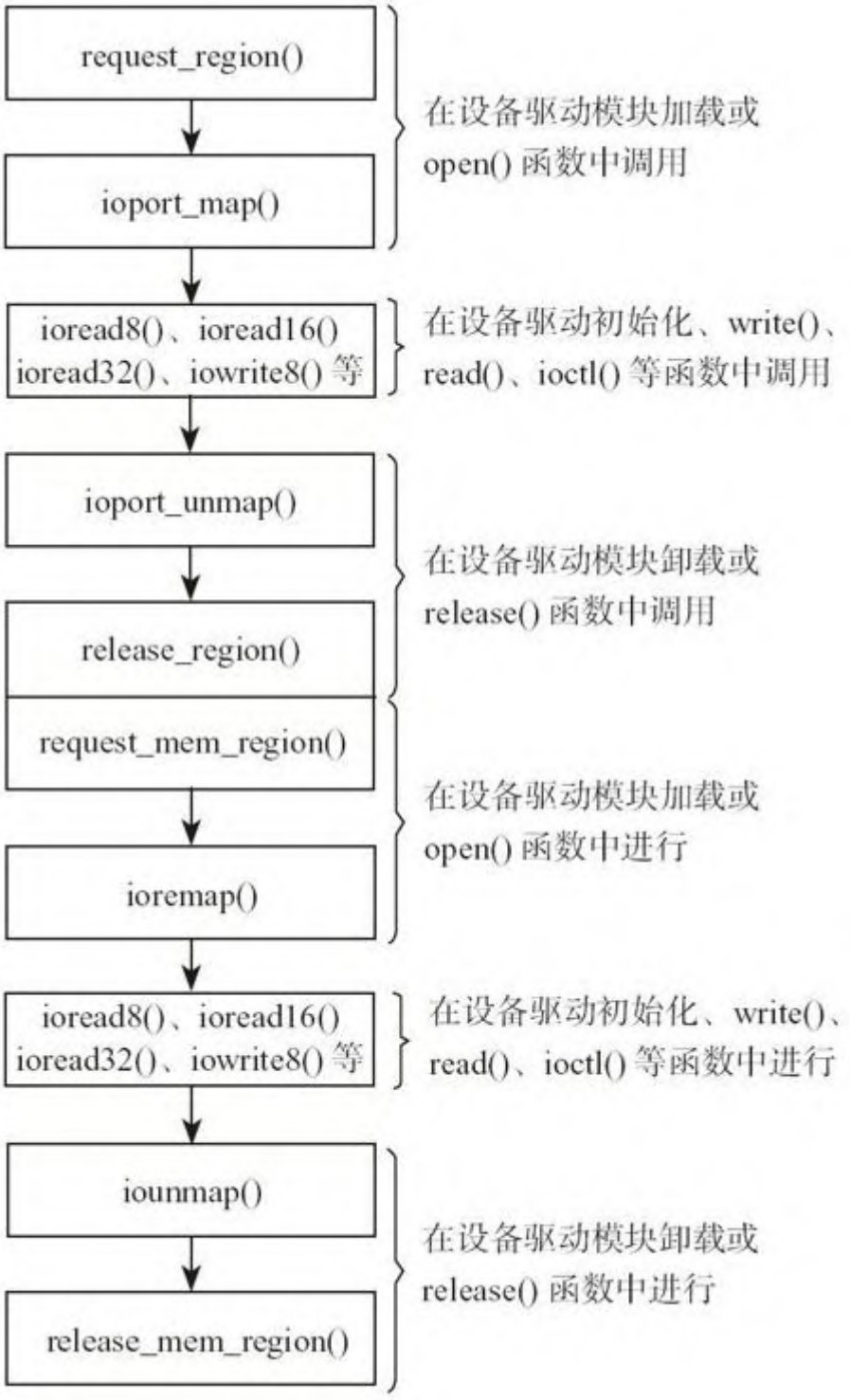


图7-8　I/O端口的访问流程（映射到内存空间）

1.内存映射与 VMA

一般情况下，用户空间是不可能也不应该直接访问设备的，但是，设备驱动程序中可实现 mmap（）函数，这个函数可使得用户空间直接访问设备的物理地址。实际上， mmap（）实现了这样的一个映射过程：它将用户空间的一段内存与设备内存关联，当用户访问用户空间的这段地址范围时，实际上会转化为对设备的访问。

这个功能对于显示适配器一类的设备非常有意义，如果用户空间可直接通过内存映射访问显存的话，屏幕帧的各点的像素将不再需要从用户空间到内核空间的复制过程。

mmap（）必须以 PAGE_SIZE为单位进行映射，实际上，内存只能以页为单位进行映射，若要映射非 PAGE_SIZE整数倍的地址范围，要先进行页对齐，强行以 PAGE_SIZE的倍数大小进行映射。

mmap（）函数是 file_operations文件操作结构体的一个成员，其函数原型如下：

```
int（*mmap）（struct file*，struct vm_area_struct*）；
```

驱动中的 mmap（）函数将在用户进行 mmap（）系统调用时最终被调用， mmap（）系统调用的原型与 file_operations中 mmap（）的原型区别很大，如下所示：

```
caddr_t mmap（caddr_t addr，size_t len，int prot，int flags，int fd，off_t offset）；
```

参数 fd为文件描述符，一般由 open（）返回， fd也可以指定为 -1，此时须指定 flags参数中的 MAP_ANON，表明进行的是匿名映射。

len是映射到调用用户空间的字节数，它从被映射文件开头 offset个字节开始算起， offset参数一般设为 0，表示从文件头开始映射。

prot参数指定访问权限，可取如下几个值的“或”： PROT_READ（可读）、 PROT_WRITE（可写）、 PROT_EXEC（可执行）和

PROT_NONE（不可访问）。

参数 addr指定文件应被映射到用户空间的起始地址，一般指定为NULL，这样，选择起始地址的任务将由内核完成，而函数的返回值就是映射到用户空间的地址。其类型 caddr_t实际上就是 void*。

当用户调用 mmap（）的时候，内核会进行如下处理。

1）在进程的虚拟空间查找一块 VMA。

2）将这块 VMA进行映射。

3）如果设备驱动程序或者文件系统的 file_operations定义了mmap（）操作，则调用它。

4）将这个 VMA插入进程的 VMA链表中。

file_operations中 mmap（）函数的第一个参数就是步骤 1）中找到的 VMA。

由 mmap（）系统调用映射的内存可由 munmap（）解除映射，这个函数的原型如下：

```
int munmap（caddr_t addr，size_t len）；
```

驱动程序中 mmap（）的实现机制是建立页表，并填充 VMA结构体中vm_operations_struct指针。 VMA即 vm_area_struct，用于描述一个虚拟内存区域， VMA结构体的定义如代码清单 7-7所示。

代码清单 7-7　VMA结构体

```
struct vm_area_struct
{
    struct mm_struct *vm_mm; /* 所处的地址空间 */
    unsigned long vm_start; /* 开始虚拟地址 */
```

```
    unsigned long vm_end; /* 结束虚拟地址 */

    pgprot_t vm_page_prot; /* 访问权限 */
    unsigned long vm_flags; /* 标志 */
    ...
    /* 操作VMA的函数集指针 */
    struct vm_operations_struct *vm_ops;

    unsigned long vm_pgoff; /* 偏移（页帧号） */
    struct file *vm_file;
    void *vm_private_data;
    ...
};
```

VMA结构体描述的虚拟地址介于 vm_start和 vm_end之间，而其vm_ops成员指向这个 VMA的操作集。针对 VMA的操作都被包含在vm_operations_struct结构体中， vm_operations_struct结构体的定义如代码清单 7-8所示。

代码清单 7-8　vm_operations_struct结构体

```
struct vm_operations_struct
{
    void(*open)(struct vm_area_struct *area); /*打开VMA的函数*/
    void(*close)(struct vm_area_struct *area); /*关闭VMA的函数*/
    struct page *(*nopage)(struct vm_area_struct *area,
unsigned long address, int *type); /*访问的页不在内存时调用*/
    int(*populate)(struct vm_area_struct *area, unsigned long
address,unsigned long len, pgprot_t prot, unsigned long pgoff,
int nonblock);
    ...
};
```

在内核生成一个 VMA后，它会调用该 VMA的 open（）函数，例如新生成（ fork）一个继承父资源的子进程时。但是，当用户进行mmap（）系统调用后，尽管 VMA在设备驱动文件操作结构体的mmap（）被调用前就已产生，内核却不会调用 VMA的 open（）函数，通常需要在设备驱动的 mmap（）函数中显性调用 vma->vm_ops-

>open（）。代码清单 7-9给出了一个 vm_operations_struct的操作范例。

代码清单 7-9　vm_operations_struct操作范例

```
1 static int xxx_mmap(struct file *filp, struct vm_area_struct
*vma)
2 {
3     if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
vma->vm_end - vma
4 ->vm_start, vma->vm_page_prot)) /* 建立页表 */
5         return - EAGAIN;
6     vma->vm_ops = &xxx_remap_vm_ops;
7     xxx_vma_open(vma);
8     return 0;
9 }
10
11 void xxx_vma_open(struct vm_area_struct *vma) //VMA打开函数
12 {
13     ...
14     printk(KERN_NOTICE "xxx VMA open, virt %lx, phys
%lx\n",vma->vm_start,
15     vma->vm_pgoff << PAGE_SHIFT);
16 }
17
18 void xxx_vma_close(struct vm_area_struct *vma) //VMA关闭函数
19 {
20     ...
21     printk(KERN_NOTICE "xxx VMA close.\n");
22 }
23
24 static struct vm_operations_struct xxx_remap_vm_ops = //VMA
操作结构体
25 {
26     .open = xxx_vma_open,
27     .close = xxx_vma_close,
28     ...
29 };
```

第 3行调用的 remap_pfn_range（）创建页表，以 VMA结构体的成员（ VMA的数据成员是内核根据用户的请求自己填充的）作为

remap_pfn_range（）的参数，映射的虚拟地址范围是 vma->vm_start 至 vma->vm_end。

remap_pfn_range（）函数的原型如下：

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long
addr,unsigned long pfn, unsigned long size, pgprot_t prot);
```

其中 addr参数表示内存映射开始处的虚拟地址。
remap_pfn_range（）函数为 addr～ addr+size之间的虚拟地址构造页表。

pfn是虚拟地址应该映射到的物理地址的页帧号，实际上就是物理地址右移 PAGE_SHIFT位。若 PAGE_SIZE为 4KB，则 PAGE_SHIFT为 12，因为 PAGE_SIZE等于 1<<PAGE_SHIFT。

prot是新页所要求的保护属性。

在驱动程序中，我们能使用 remap_pfn_range（）映射内存中的保留页（如 x86系统中的 640KB～ 1MB区域）和设备 I/O内存，另外，kmalloc（）申请的内存若要被映射到用户空间，可以通过 mem_map_reserve（）设置为保留后进行。代码清单 7-10给出了映射 kmalloc（）申请的内存到用户空间的典型范例。

代码清单 7-10 映射 kmalloc（）申请的内存到用户空间范例

```
1 /*内核模块加载函数*/
2 int __init kmalloc_map_init(void)
3 {
4     ...//申请设备号、添加cdev结构体
5     buffer = kmalloc(BUF_SIZE, GFP_KERNEL); //申请buffer
6
7     for (page = virt_to_page(buffer); page <
virt_to_page(buffer + BUF_SIZE);
8 page++)
9     {
```

```
10          mem_map_reserve(page); //置页为保留
11      }
12 }
13 /*mmap()函数*/
14 static int kmalloc_map_mmap(struct file *filp, struct
vm_area_struct *vma)
15 {
16      unsigned long page, pos;
17      unsigned long start = (unsigned long)vma->vm_start;
18      unsigned long size = (unsigned long)(vma->vm_end - vma-
>vm_start);
19      printk(KERN_INFO "mmaptest_mmap called\n");
20      /* 用户要映射的区域太大 */
21      if (size > BUF_SIZE)
22          return - EINVAL;
23
24      pos = (unsigned long)buffer;
25      /* 映射buffer中的所有页 */
26      while (size > 0)
27      {
28          /* 每次映射一页 */
29          page = virt_to_phys((void*)pos);
30          if (remap_page_range(start, page, PAGE_SIZE,
PAGE_SHARED))
31              return - EAGAIN;
32          start += PAGE_SIZE;
33          pos += PAGE_SIZE;
34          size -= PAGE_SIZE;
35      } return 0;
36 }
```

第 30行调用 remap_page_range（ start， page， PAGE_SIZE，PAGE_SHARED）的第 4个参数 PAGE_SHARED实际上是 _PAGE_PRESENT|_PAGE_USER|_PAGE_RW，表明可读写并映射到用户空间。

通常， I/O内存被映射时需要是不允许高速缓存（ nocache）的，这时候我们应该对 vma->vm_page_prot设置 nocache标志之后再映射，如代码清单 7-11所示。

代码清单 7-11　以 nocache方式将内核空间映射到用户空间

```
1 static int xxx_nocache_mmap(struct file *filp, struct
vm_area_struct *vma)
2 {
3     vma->vm_page_prot = pgprot_noncached(vma-
>vm_page_prot);// 赋nocache标志
4     vma->vm_pgoff = ((u32)map_start >> PAGE_SHIFT);
5     //映射
6     if (remap_pfn_range(vma, vma->vm_start, vma-
>vm_pgoff,vma->vm_end - vma
7 ->vm_start, vma->vm_page_prot))
8     return - EAGAIN;
9     return 0;
10 }
```

上述代码第 3行的 pgprot_noncached（）是一个宏，它高度依赖于CPU体系结构， ARM的 pgprot_noncached（）定义如下：

#define pgprot_noncached（ prot） __pgprot（ pgprot_val（ prot） &~（ L_PTE_CACHEABLE|L_PTE_BUFFERABLE））

另一个比 pgprot_noncached（）稍微少一些限制的宏是 pgprot_writecombine（），它的定义如下：

```
#define pgprot_noncached(prot) __pgprot(pgprot_val(prot) & ~
(L_PTE_CACHEABLE | L_PTE_ BUFFERABLE))
```

---

pgprot_noncached（）实际禁止了相关页的 Cache和写缓冲（ write buffer）， pgprot_writecombine（）则没有禁止写缓冲。 ARM的写缓冲器是一个非常小的 FIFO存储器，位于处理器核与内存之间，其目的在于将处理器核和 Cache从较慢的内存写操作中解脱出来。写缓冲区与 Cache在存储层次上处于同一层次，但是它只作用于写内存。

2.nopage（）函数

除了 remap_pfn_range（）以外，在驱动程序中实现 VMA的 nopage（）函数通常可以为设备提供更加灵活的内存映射途径。当访

问的页不在内存，即发生缺页异常时， nopage（）会被内核自动调用。这是因为，当发生缺页异常时系统会经过如下处理过程：

1）找到缺页的虚拟地址所在的 VMA。

2）如果必要，分配中间页目录表和页表。

3）如果页表项对应的物理页面不存在，则调用这个 VMA的 nopage（）方法，后者返回物理页面的页描述符。

4）将物理页面的地址填充到页表中。

实现 nopage（）后，用户空间可以通过 mremap（）系统调用重新绑定映射区域所绑定的地址，代码清单 7-12给出了一个设备驱动中使用 nopage（）的典型范例。

代码清单 7-12　nopage（）函数使用范例

```
static int xxx_mmap(struct file *filp, struct vm_area_struct
*vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    if (offset >= __pa(high_memory) || (filp->f_flags &O_SYNC))
         vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED; //预留
    vma->vm_ops = &xxx_nopage_vm_ops;
    xxx_vma_open(vma);
    return 0;
}

struct page *xxx_vma_nopage(struct vm_area_struct *vma,
unsigned long address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
/* 物理地址 */
    unsigned long pageframe = physaddr >> PAGE_SHIFT; /* 页帧号
*/
    if (!pfn_valid(pageframe)) /* 页帧号有效？ */
        return NOPAGE_SIGBUS;
```

```
    pageptr = pfn_to_page(pageframe); /* 页帧号->页描述符 */
    get_page(pageptr); /* 获得页,增加页的使用计数 */
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr; /*返回页描述符 */
}
```

上述函数对常规内存进行映射，返回一个页描述符，可用于扩大或缩小映射的内存区域。

由此可见， nopage（）与 remap_pfn_range（）的一个较大区别在于 remap_pfn_range（）一般用于设备内存映射，而 nopage（）还可用于 RAM映射。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

7.5 DMA

DMA是一种无须 CPU的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。使用 DMA可以使系统 CPU从实际的 I/O数据传输过程中摆脱出来，从而大大提高系统的吞吐率。 DMA通常与硬件体系结构特别是外设的总线技术密切相关。

DMA方式的数据传输由 DMA控制器（ DMAC）控制，在传输期间， CPU可以并发地执行其他任务。当 DMA结束后， DMAC通过中断通知 CPU数据传输已经结束，然后由 CPU执行相应的中断服务程序进行后处理。

7.5.1 DMA与 Cache的一致性

Cache和 DMA本身似乎是两个毫不相关的事物。 Cache被用做 CPU针对内存的缓存，利用程序的空间局部性和时间局部性原理，达到较高的命中率从而避免 CPU每次都一定要与相对慢速的内存交互数据来提高数据的访问速率。 DMA可以用做内存与外设之间传输数据的方式，这种传输方式之下，数据并不需要经过 CPU中转。

假设 DMA针对内存的目的地址与 Cache缓存的对象没有重叠区域，DMA和 Cache之间将相安无事。但是，如果 DMA的目的地址与 Cache所缓存的内存地址访问有重叠，经过 DMA操作， Cache缓存对应的内存数据已经被修改，而 CPU本身并不知道，它仍然认为 Cache中的数据就是内存中的数据，以后访问 Cache映射的内存时，它仍然使用陈旧的 Cache数据。这样就发生 Cache与内存之间数据“不一致性”的错误。

所谓 Cache数据与内存数据的不一致性，是指在采用 Cache的系统中，同样一个数据可能既存在于 Cache中，也存在于内存中， Cache与内存中的数据一样则具有一致性，数据若不一样则具有不一致性。

需要特别注意的是， Cache与内存的一致性问题经常被初学者遗忘。在发生 Cache与内存不一致性错误后，驱动将无法正常运行。如果没有相关的背景知识，工程师几乎无法定位错误的原因，因为看起来所有的程序都是完全正确的。

解决由于 DMA导致的 Cache一致性问题的最简单方法是直接禁止 DMA目标地址范围内内存的 Cache功能。当然，这将牺牲性能，但是却更可靠。

7.5.2　Linux下的 DMA编程

这里要提醒大家注意的是， DMA本身不属于字符设备、块设备和网络设备中的任何一种，它只是这些外设与内存交互数据的一种硬件手段。因此，本节的标题不是“ Linux下的 DMA驱动”而是“ Linux下的 DMA编程”。由于经常有人问我“ DMA驱动”的问题，所以这里特别澄清，希望读者不要再在 DMA相关的程序中寻找驱动的架构代码。

内存中用于与外设交互数据的一块区域被称做 DMA缓冲区，在设备不支持 scatter/gather（分散 /聚集）操作的情况下， DMA缓冲区必须是物理上连续的。对于 ISA设备而言，其 DMA操作只能在 16MB以下的内存中进行，因此，在使用 kmalloc（）和 __get_free_pages（）及其类似函数申请 DMA缓冲区时应使用 GFP_DMA标志，这样能保证获得的内存是具备 DMA能力的（ DMA-capable）。

内核中定义了 __get_free_pages（）针对 DMA的“快捷方式” __get_dma_pages（），它在申请标志中添加了 GFP_DMA，如下所示：

```
#define _ _get_dma_pages(gfp_mask, order) \
_ _get_free_pages((gfp_mask) | GFP_DMA,(order))
```

如果不想使用 log2size即 order为参数申请 DMA内存，则可以使用另一个函数 dma_mem_alloc（），其源代码如代码清单 7-13所示。

代码清单 7-13　dma_mem_alloc（）函数

```
static unsigned long dma_mem_alloc(int size)
{
    int order = get_order(size);//大小->指数
    return _ _get_dma_pages(GFP_KERNEL, order);
}
```

基于 DMA的硬件使用总线地址而非物理地址，总线地址是从设备角度上看到的内存地址，物理地址则是从 CPU角度上看到的未经转换的内存地址（经过转换的为虚拟地址）。虽然在 PC上，对于 ISA和 PCI而言，总线地址即为物理地址，但并非每个平台都是如此。因为有时候接口总线通过桥接电路连接，桥接电路会将 I/O地址映射为不同的物理地址。例如，在 PReP（ PowerPC Reference Platform）系统中，物理地址 0在设备端看起来是 0x80000000，而 0通常又被映射为虚拟地址 0xC0000000，所以同一地址就具备了三重身份：物理地址 0、总线地址 0x80000000及虚拟地址 0xC0000000。还有一些系统提供了页面映射机制，它能将任意的页面映射为连续的外设总线地址。内核提供了如下函数用于进行简单的虚拟地址 /总线地址转换：

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

在使用 IOMMU或反弹缓冲区的情况下，上述函数一般不会正常工作。而且，不建议使用这两个函数。如图 7-9所示， IOMMU的工作原理与 CPU内的 MMU非常类似，不过它针对的是外设总线地址和内存地址之间的转化。由于 IOMMU可以使得外设看到“虚拟地址”，因此在使用 IOMMU的情况下，在修改映射寄存器后，可以使得 SG中分段的缓冲区地址对外设变得连续。

注： SG是指 Scatter-Gather DMA方式，与其对应的就是 Block DMA方式。 DMA传输中要求源物理地址和目标物理地址必须是连续的。因此 DMA初始设计时支持的是 Block DMA传输方式，即传输完物理上连续的数据后， DMA控制器会向 CPU发送中断，以请求下一块物理上连续的数据传输。但连续的物理空间在计算机系统中是宝贵资源，于是人们又设计出了 Scatter-Gather DMA，它会使用一个链表来描述物理上不连续的内存空间，然后把链表首地址告诉 DMA控制器， DMA控制器在传输完一块链表元素所指定的物理连续数据后，并不马上发出中断，而从链表取下一元素以指示下一次物理连续的数据传输，直至链表尾才发送中断。明显 SG方式并不再要求 DMA传输前先要准备好一整块大的物理连续内存空间，它可以把杂散的小物理空间整合成一块虚拟的大块连续物理空间。显然 Scatter-Gather DMA方式比 Block DMA

方式效率更高。在后面我们还会看到以 sg打头的函数，它们就与Scatter-Gather DMA方式有关。

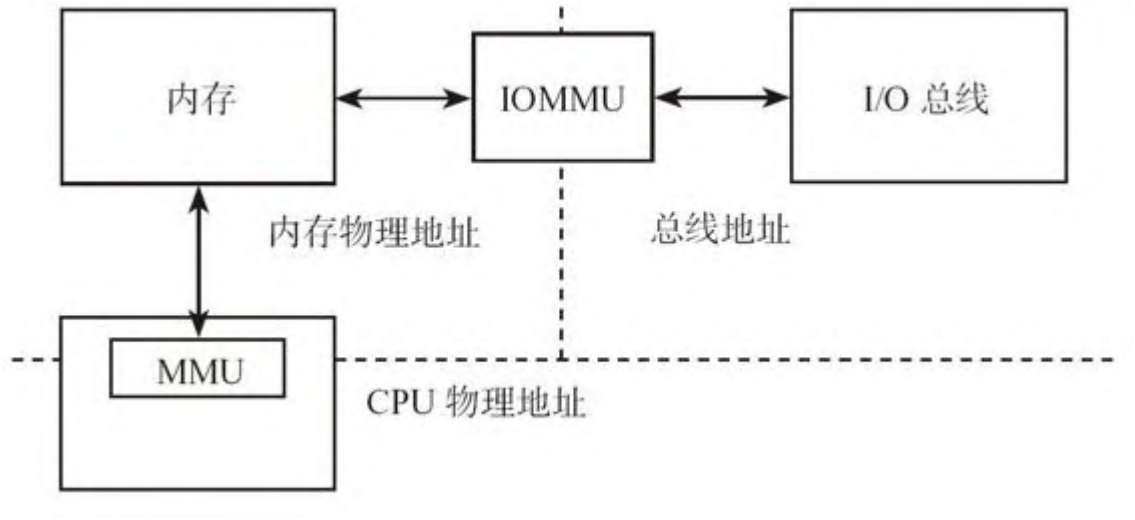


图7-9　MMU与IOMMU

设备并不一定能在所有的内存地址上执行 DMA操作，在这种情况下应该通过下列函数执行 DMA地址掩码：

```
int dma_set_mask（struct device*dev，u64mask）；
```

例如，对于只能在 24位地址上执行 DMA操作的设备而言，就应该调用dma_set_mask（ dev， 0xffffff）。

DMA映射包括两个方面的工作：分配一片 DMA缓冲区；为这片缓冲区产生设备可访问的地址。同时， DMA映射也必须考虑 Cache一致性问题。内核中提供了以下函数用于分配一个 DMA一致性的内存区域：

```
void*dma_alloc_coherent（struct device*dev，size_t size，
dma_addr_t*handle，gfp_t gfp）；
```

上述函数的返回值为申请到的 DMA缓冲区的虚拟地址，此外，该函数还通过参数 handle返回 DMA缓冲区的总线地址。 handle的类型为dma_addr_t，代表的是总线地址。

dma_alloc_coherent（）申请一片 DMA缓冲区，进行地址映射并保证该缓冲区的 Cache一致性。与 dma_alloc_coherent（）对应的释放函数为：

```
void dma_free_coherent（struct device*dev，size_t size，
void*cpu_addr，dma_addr_t handle）；
```

以下函数用于分配一个写合并（ writecombining）的 DMA缓冲区：

```
void*dma_alloc_writecombine（struct device*dev，size_t size，
dma_addr_t*handle，gfp_t gfp）；
```

与 dma_alloc_writecombine（）对应的释放函数 dma_free_writecombine（）实际上就是 dma_free_coherent（），因为它定义为：

```
#define dma_free_writecombine(dev,size,cpu_addr,handle) \
dma_free_coherent(dev,size,cpu_addr,handle)
```

此外， Linux内核还提供了 PCI设备申请 DMA缓冲区的函数 pci_alloc_consistent（），其原型为：

```
void*pci_alloc_consistent（struct pci_dev*pdev，size_t size，
dma_addr_t*dma_addrp）；
```

对应的释放函数为 pci_free_consistent（），其原型为：

```
void pci_free_consistent（struct pci_dev*pdev，size_t size，
void*cpu_addr，dma_addr_t dma_addr）；
```

相对于一致性 DMA映射而言，流式 DMA映射的接口较为复杂。对于单个已经分配的缓冲区而言，使用 dma_map_single（）可实现流式 DMA映射，该函数原型为：

```
dma_addr_t dma_map_single（struct device*dev，void*buffer，
size_t size，enum dma_data_direction direction）；
```

如果映射成功，返回的是总线地址，否则返回 NULL。第 4个参数为DMA的方向，可能的值包括 DMA_TO_DEVICE、 DMA_FROM_DEVICE、DMA_BIDIRECTIONAL和 DMA_NONE。

dma_map_single（）的反函数为 dma_unmap_single（），原型是：

```
void dma_unmap_single（struct device*dev，dma_addr_t dma_addr，
size_t size，enum dma_data_direction direction）；
```

通常情况下，设备驱动不应该访问没有映射好的流式 DMA缓冲区，如果一定要这么做，可先使用如下函数获得 DMA缓冲区的拥有权：

```
void dma_sync_single_for_cpu（struct device*dev，dma_handle_t
bus_addr，size_t size，enum dma_data_direction direction）；
```

在驱动访问完 DMA缓冲区后，应该将其所有权返还给设备，可通过如下函数完成：

```
void dma_sync_single_for_device（struct device*dev，dma_handle_t bus_addr，size_t size，enum dma_data_direction direction）；
```

如果设备要求较大的 DMA缓冲区，在其支持 SG模式的情况下，申请多个不连续的、相对较小的 DMA缓冲区通常是防止申请太大的连续物理空间的方法。在 Linux内核中，使用如下函数映射 SG：

```
int dma_map_sg（struct device*dev，struct scatterlist*sg，int nents，enum dma_data_direction direction）；
```

nents是散列表（ scatterlist）入口的数量，该函数的返回值是 DMA缓冲区的数量，可能小于 nents。对于 scatterlist中的每个项目，dma_map_sg（）为设备产生恰当的总线地址，它会合并物理上临近的内存区域。

scatterlist结构体的定义如代码清单 7-14所示，它包含了 scatterlist对应的 page结构体指针、缓冲区在 page中的偏移（ offset）、缓冲区长度（ length）以及总线地址（ dma_address）。

代码清单 7-14　scatterlist结构体

```
struct scatterlist
{
    struct page *page;
    unsigned int offset;
    dma_addr_t dma_address;
    unsigned int length;
};
```

执行 dma_map_sg（）后，通过 sg_dma_address（）可返回 scatterlist对应缓冲区的总线地址， sg_dma_len（）可返回 scatterlist对应缓冲区的长度，这两个函数的原型为：

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

在 DMA传输结束后，可通过 dma_map_sg（）的反函数 dma_unmap_sg（）去除 DMA映射：

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
int nents, enum dma_data_direction direction);
```

SG映射属于流式 DMA映射，与单一缓冲区情况下的流式 DMA映射类似，如果设备驱动一定要访问映射情况下的 SG缓冲区，应该先调用如下函数：

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist
*sg, int nents, enum dma_data_direction direction);
```

访问完后，通过下列函数将所有权返回给设备：

```
void dma_sync_sg_for_device(struct device *dev, struct
scatterlist *sg, int nents, enum dma_data_direction direction);
```

Linux系统中可以有一个相对简单的方法预先分配缓冲区，那就是同步"mem="参数预留内存。例如，对于内存为 64MB的系统，通过给其传递mem=62MB命令行参数可以使得顶部的 2MB内存被预留出来作为 I/O内存使用，这 2MB内存可以被静态映射（见 7.4.4节），也可以被执行ioremap（）。

1.申请和释放DMA通道

与中断一样，在使用 DMA之前，设备驱动程序需首先向系统申请 DMA通道，申请 DMA通道的函数如下：

```
int request_dma（unsigned int dmanr，const char*device_id）；
```

同样，设备结构体指针可作为传入 device_id的最佳参数。使用完DMA通道后，应该利用如下函数释放该通道：

```
void free_dma（unsigned int dmanr）；
```

现在可以总结出在 Linux设备驱动中 DMA相关代码的流程，如图 7-10所示。

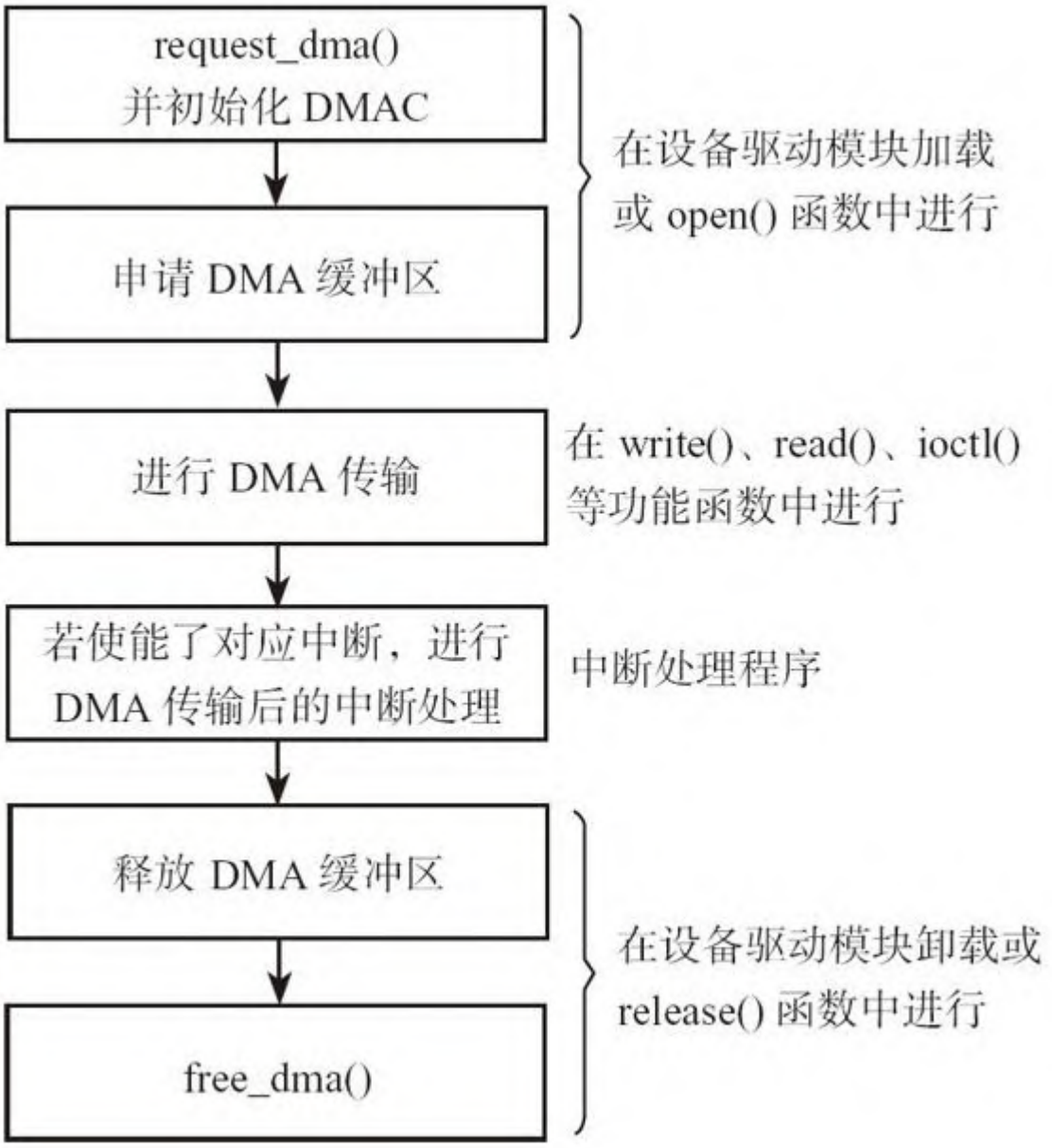


图7-10　Linux中DMA使用流程

2.DMA控制器使用实例

代码清单 7-15给出的一组函数可以从硬件上设置某 DMA控制器，包括使能和禁止 DMA通道，设置 DMA的模式、地址和尺寸等。 DMA flip-flop用来控制对 16位寄存器（由两个 8位字节组成）的访问，清除时选中低字节，设置时访问高字节。

代码清单 7-15　某 DMA硬件设置

```
1 /*使能DMA通道*/
2 static _ _inline_ _ void enable_dma(unsigned int dmanr)
3 {
4 if (dmanr <= 3)
5 dma_outb(dmanr, DMA1_MASK_REG);
6 else
7 dma_outb(dmanr &3, DMA2_MASK_REG);
8 }
9
10 /*禁止DMA通道*/
11 static _ _inline_ _ void disable_dma(unsigned int dmanr)
12 {
13 if (dmanr <= 3)
14 dma_outb(dmanr | 4, DMA1_MASK_REG);
15 else
16 dma_outb((dmanr &3) | 4, DMA2_MASK_REG);
17 }
18
19 /* 设置传输尺寸(通道0~3最大64KB, 通道5~7 最大128KB */
20 static _ _inline_ _ void set_dma_count(unsigned int dmanr,
unsigned int count)
21 {
22 count--;
23 if (dmanr <= 3)
24 {
25 dma_outb(count &0xff, ((dmanr &3) << 1) + 1+IO_DMA1_BASE);
26 dma_outb((count >> 8) &0xff, ((dmanr &3) << 1) +
1+IO_DMA1_BASE);
27 }
28 else
29 {
30 dma_outb((count >> 1) &0xff, ((dmanr &3) << 2) +
2+IO_DMA2_BASE);
31 dma_outb((count >> 9) &0xff, ((dmanr &3) << 2) +
2+IO_DMA2_BASE);
32 }
33 }
34
35 /* 设置传输地址和页位 */
36 static _ _inline_ _ void set_dma_addr(unsigned int dmanr,
unsigned
int a)
37 {
```

```
38 set_dma_page(dmanr, a >> 16);
39 if (dmanr <= 3)
40 {
41 dma_outb(a &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
42 dma_outb((a >> 8) &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
43 }
44 else
45 {
46 dma_outb((a >> 1) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
47 dma_outb((a >> 9) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
48 }
49 }
50
51 /* 设置页寄存器位,当已知DMA当前地址寄存器的低16位时,连续传输 */
52 static _ _inline_ _ void set_dma_page(unsigned int dmanr,
char pagenr)
53 {
54 switch (dmanr)
55 {
56 case 0:
57 dma_outb(pagenr, DMA_PAGE_0);
58 break;
59 case 1:
60 dma_outb(pagenr, DMA_PAGE_1);
61 break;
62 case 2:
63 dma_outb(pagenr, DMA_PAGE_2);
64 break;
65 case 3:
66 dma_outb(pagenr, DMA_PAGE_3);
67 break;
68 case 5:
69 dma_outb(pagenr &0xfe, DMA_PAGE_5);
70 break;
71 case 6:
72 dma_outb(pagenr &0xfe, DMA_PAGE_6);
73 break;
74 case 7:
75 dma_outb(pagenr &0xfe, DMA_PAGE_7);
76 break;
77 }
78 }
79
80 /*清除DMA flip-flop*/
81 static _ _inline_ _ void clear_dma_ff(unsigned int dmanr)
82 {
83 if (dmanr <= 3)
```

```
84 dma_outb(0, DMA1_CLEAR_FF_REG);
85 else
86 dma_outb(0, DMA2_CLEAR_FF_REG);
87 }
88
89 /*设置某通道的DMA模式*/
90 static _ _inline_ _ void set_dma_mode(unsigned int dmanr,
char mode)
91 {
92 if (dmanr <= 3)
93 dma_outb(mode | dmanr, DMA1_MODE_REG);
94 else
95 dma_outb(mode | (dmanr &3), DMA2_MODE_REG);
96 }
```

如设备 xxx使用了 DMA， DMA相关的信息应该被添加到设备结构体内。在模块加载函数或打开函数中，应该申请 DMA通道和中断，而初始化 DMA本身。不论是内存到 I/O的 DMA，还是 I/O到内存的 DMA，都应使用代码清单 7-15中的硬件设置函数。中断服务程序完成 DMA的善后处理，有时候正是中断事件的到来才触发了一次 DMA的传输。

内存到 I/O的 DMA发送通常由上层触发，而 I/O到内存的 DMA传送通常是由外设收到了数据之后的中断触发。代码清单 7-16给出了外设使用某 DMA控制器进行数据传输的设备驱动典型范例。

代码清单 7-16　外设使用某 DMA驱动范例

```
1 /*xxx设备结构体*/
2 typedef struct
3 {
4     ...
5     void *dma_buffer; //DMA缓冲区
6     /*当前DMA的相关信息*/
7     struct
8     {
9         unsigned int direction; //方向
10        unsigned int length; //尺寸
11        void *target; //目标
12        unsigned long start_time; //开始时间
13     } current_dma;
```

```
14
15      unsigned char dma;//DMA通道
16  } xxx_device;
17
18  static int xxx_open(...)
19  {
20      ...
21      /*安装中断服务程序 */
22      if ((retval = request_irq(dev->irq, &xxx_interrupt, 0,
dev->name,dev))) {
23          printk(KERN_ERR "%s: could not allocate IRQ%d\n",
dev->name,dev->irq);
24              return retval;
25      }
26      /*申请DMA*/
27      if ((retval = request_dma(dev->dma, dev->name))) {
28          free_irq(dev->irq, dev);
29          printk(KERN_ERR "%s: could not allocate DMA%d
channel\n", ...);
30          return retval;
31      }
32      /*申请DMA缓冲区*/
33      dev->dma_buffer = (void *)
dma_mem_alloc(DMA_BUFFER_SIZE);
34      if (!dev->dma_buffer) {
35          printk(KERN_ERR "%s: could not allocate DMA
buffer\n",dev->name);
36          free_dma(dev->dma);
37          free_irq(dev->irq, dev);
38          return -ENOMEM;
39      }
40      /*初始化DMA*/
41      init_dma();
42      ...
43  }
44
45  /*内存到外设*/
46  static int mem_to_xxx(const byte *buf,int len)
47  {
48      ...
49      dev->current_dma.direction = 1; /*DMA方向*/
50      dev->current_dma.start_time = jiffies; /*记录DMA开始时间*/
51
52      memcpy(dev->dma_buffer, buf, len); /*复制要发送的数据到DMA
缓冲区*/
53      target = isa_virt_to_bus(dev->dma_buffer);/*假设xxx 挂接在
```

```
ISA 总线*/
54
55      /*进行一次DMA写操作*/
56      flags=claim_dma_lock();
57      disable_dma(dev->dma); /*禁止DMA*/
58      clear_dma_ff(dev->dma); /*清除DMA flip-flop*/
59      set_dma_mode(dev->dma, 0x48); /* DMA 内存 -> io */
60      set_dma_addr(dev->dma, target); /*设置DMA地址*/
61      set_dma_count(dev->dma, len); /*设置DMA长度*/
62      outb_control(dev->x_ctrl | DMAE | TCEN, dev);/*让设备接收
DMA*/
63      enable_dma(dev->dma); /*使能DMA*/
64      release_dma_lock(flags);
65
66      printk(KERN_DEBUG "%s: DMA transfer started\n", dev-
>name);
67      ...
68 }
69
70 /*外设到内存*/
71 static void xxx_to_mem(const char *buf, int len,char *
target)
72 {
73      ...
74      /*记录DMA信息*/
75      dev->current_dma.target = target;
76      dev->current_dma.direction = 0;
77      dev->current_dma.start_time = jiffies;
78      dev->current_dma.length = len;
79
80      /*进行一次DMA读操作*/
81      outb_control(dev->x_ctrl | DIR | TCEN | DMAE, dev);
82      flags = claim_dma_lock();
83      disable_dma(dev->dma);
84      clear_dma_ff(dev->dma);
85      set_dma_mode(dev->dma, 0x04); /* I/O->mem */
86      set_dma_addr(dev->dma, isa_virt_to_bus(target));
87      set_dma_count(dev->dma, len);
88      enable_dma(dev->dma);
89      release_dma_lock(flags);
90      ...
91 }
92
93 /*设备中断处理*/
94 static irqreturn_t xxx_interrupt(int irq, void *dev_id,
struct pt_regs *reg_ptr)
```

```
95 {
96     ...
97     do
98     {
99         /* DMA传输完成? */
100         if (int_type==DMA_DONE)
101         {
102             outb_control(dev->x_ctrl &~(DMAE | TCEN | DIR),
dev);
103             if (dev->current_dma.direction)
104             {
105                 /*内存->I/O*/
106                 ...
107             }
108             else
109                 /*I/O->内存*/
110             {
111             memcpy(dev->current_dma.target, dev-
>dma_buffer, dev
112 ->current_dma.length);
113         }
114     }
115 else if(int_type==RECV_DATA) /*收到数据*/
116 {
117         xxx_to_mem(...);/*通过DMA读接收到的数据到内存*/
118 }
119 ...
120 }
121 ...
122 }
```

第8章　Linux设备驱动中的中断

在访问设备时，如果不管设备是否有数据都死等它的数据，那别的设备就得不到访问。因此，在硬件设计中我们引进了中断机制。

本章主要讲解Linux设备驱动编程中的中断处理。中断服务程序的执行是与一般的进程异步的，也就是不存在于进程上下文，所以要求中断服务程序的时间尽可能短。为此，Linux在中断处理中引入了顶半部和底半部分离机制。另外，内核中对时钟的处理也采用中断方式，而内核软件定时器最终依赖于时钟中断。

8.1 Linux中断及中断处理架构

设备中断的到来会打断内核中进程的正常调度和运行，因此，系统更高吞吐率的追求导致要求中断服务程序尽可能的短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大量的耗时处理。

为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点， Linux将中断处理程序分解为两个半部：顶半部（ top half）和底半部（ bottom half）。

顶半部完成尽可能少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态并清除中断标志后就进行“登记中断”的工作。所谓“登记中断”就是指，将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，从而使得 CPU能够去服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，由它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部则相对来说并不是非常紧急的，而且相对比较耗时，因此不便在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，由此认为 Linux设备驱动中的中断处理一定要分为两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。

在 Linux系统中，查看 /proc/interrupts文件可以获得系统中断的统计信息，如下所示。在单处理器的系统中，第一列是中断号，第二列是向 CPU0产生该中断的次数，之后的是对于中断的描述。

```
#cat /proc/interrupts
           CPU0
1:            1    goldfish  goldfish_pdev_bus
3:        31865    goldfish  Goldfish Timer Tick
```

```
4:          0    goldfish  goldfish_tty
10:         0    goldfish  goldfish_rtc
11:         0    goldfish  goldfish_tty
12:         0    goldfish  goldfish_tty
13:        98    goldfish  eth0
14:       494    goldfish  goldfish_fb
15:         0    goldfish  goldfish-battery
16:       251    goldfish  goldfish-events-keypad
17:      4183    goldfish  goldfish_pipe
18:         1    goldfish  goldfish_switch
19:         0    goldfish  goldfish_switch
Err:        0
```

8.2 Linux中断编程

中断处理与进程是 CPU上两类完全独立的执行体，因此它们有两类上下文。本节我们将讲述与中断相关的编程技术，这些技术将让我们合理地使用中断这类执行体。

8.2.1 申请和释放中断

在 Linux设备驱动中，使用中断的设备需要申请和释放对应的中断，分别使用内核提供的 request_irq（）和 free_irq（）函数。

1.申请 IRQ

要使用中断，首先要申请相应的中断资源，在硬件上就体现为关联上了某条中断线。

```
int request_irq(unsigned int irq,
void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
unsigned long irqflags,
const char * devname,
void *dev_id);
```

其中， irq是要申请的硬件中断号。

handler是向系统登记的中断处理函数，是一个回调函数，中断发生时系统调用这个函数， dev_id参数将被传递给它。

irqflags是中断处理的属性，若设置了 SA_INTERRUPT，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序则不屏蔽；若设置了 SA_SHIRQ，则表示多个设备共享中断，dev_id在中断共享时会用到，一般设置为这个设备的设备结构体或者NULL。

request_irq（）返回 0表示成功，返回 -INVAL表示中断号无效或处理函数指针为 NULL，返回 -EBUSY表示中断已经被占用且不能共享。

2.释放 IRQ

与 request_irq（）对应的函数为 free_irq（）， free_irq（）的原型如下：

```
void free_irq(unsigned int irq, void*dev_id);
```

free_irq（）中参数的定义与 request_irq（）相同。

8.2.2 使能与屏蔽中断

下列 3个函数用于屏蔽一个中断源。

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

disable_irq_nosync（）与 disable_irq（）的区别在于前者立即返回，而后者等待目前的中断处理完成。注意：这 3个函数作用于可编程中断控制器，因此，对系统内的所有 CPU都生效。

下列两个函数将屏蔽本 CPU内的所有中断。

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

前者会将目前的中断状态保留在 flags中，注意 flags被直接传递，而不是通过指针传递。后者直接禁止中断。

与上述两个禁止中断对应的恢复中断的方法如下：

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

以上各 local_开头的方法的作用范围是本 CPU内，也就是上面的中断与屏蔽，只有执行该 local_xxx_xxx（）函数的 CPU才会有相应的响应。

8.2.3 底半部机制

前面我们讲过， Linux中断的处理其实分成了上下两部分，其底半部的实现机制主要有 tasklet、工作队列和软中断。

1.tasklet

tasklet的使用较简单，我们只需要定义 tasklet及其处理函数并将两者关联，例如：

```
void my_tasklet_func(unsigned long); /*定义一个处理函数*/
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
/*定义一个tasklet结构my_tasklet,与my_tasklet_func(data)函数相关联*/
```

代码 DECLARE_TASKLET（ my_tasklet， my_tasklet_func， data）实现了定义名称为 my_tasklet的 tasklet，并将其与 my_tasklet_func（）这个函数绑定，而传入这个函数的参数为 data。

在需要调度 tasklet的时候引用一个 tasklet_schedule（）函数，这使得系统在适当的时候进行调度指定的 tasklet运行，如下所示：

```
tasklet_schedule（&my_tasklet）;
```

使用 tasklet作为底半部处理中断的设备驱动程序模板如代码清单 8-1所示（仅包含与中断相关的部分）。

代码清单 8-1　tasklet使用模板

```
1 /*定义tasklet和底半部函数并关联*/
2 void xxx_do_tasklet(unsigned long);
3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
5 /*中断处理底半部*/
6 void xxx_do_tasklet(unsigned long)
7 {
8     ...
9 }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
13 {
14     ...
15     tasklet_schedule(&xxx_tasklet);
16     ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int __init xxx_init(void)
21 {
22     ...
23     /*申请中断*/
24     result = request_irq(xxx_irq, xxx_interrupt,
25     SA_INTERRUPT, "xxx", NULL);
26     ...
27 }
28
29 /*设备驱动模块卸载函数*/
30 void __exit xxx_exit(void)
31 {
32     ...
33     /*释放中断*/
34     free_irq(xxx_irq, xxx_interrupt);
35     ...
36 }
```

上述程序在模块加载函数中申请中断（第 24～ 25行），并在模块卸载函数中释放它（第 34行）。对应于 xxx_irq的中断处理程序被设置

为 xxx_interrupt（）函数，在这个函数中，第 15行的 tasklet_schedule（ &xxx_tasklet）调度的 tasklet函数 xxx_do_tasklet（）在适当的时候得到执行。

上述代码第 12行显示中断处理程序顶半部的返回类型为 irqreturn_t，它定义为 int，中断处理程序顶半部一般返回 IRQ_HANDLED。

2.工作队列

工作队列的使用方法与 tasklet非常相似，下面的代码用于定义一个工作队列和一个底半部执行函数。

```
struct work_struct my_wq; /*定义一个工作队列*/
void my_wq_func(unsigned long); /*定义一个处理函数*/
```

通过 INIT_WORK（）可以初始化这个工作队列并将工作队列与处理函数绑定，如下所示：

```
INIT_WORK(&my_wq, (void (*)(void *)) my_wq_func, NULL);
/*初始化工作队列并将其与处理函数绑定*/
```

与 tasklet_schedule（）对应的用于调度工作队列执行的函数为 schedule_work（），如：

```
schedule_work（&my_wq）; /*调度工作队列执行*/
```

与代码清单 8-1对应，使用工作队列处理中断底半部的设备驱动程序模板，如代码清单 8-2所示（仅包含与中断相关的部分）。

代码清单 8-2　工作队列使用模板

```
1 /*定义工作队列和关联函数*/
2 struct work_struct xxx_wq;
3 void xxx_do_work(unsigned long);
4
5 /*中断处理底半部*/
6 void xxx_do_work(unsigned long)
7 {
8     ...
9 }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
13 {
14     ...
15     schedule_work(&xxx_wq);
16     ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int xxx_init(void)
21 {
22     ...
23     /*申请中断*/
24     result = request_irq(xxx_irq, xxx_interrupt,
25 SA_INTERRUPT, "xxx", NULL);
26     ...
27     /*初始化工作队列*/
28     INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work,
NULL);
29     ...
30 }
31
32 /*设备驱动模块卸载函数*/
33 void xxx_exit(void)
34 {
35     ...
36     /*释放中断*/
37     free_irq(xxx_irq, xxx_interrupt);
38     ...
39 }
```

与代码清单 8-1不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 28行）。

尽管 Linux专家们多建议在设备第一次打开时才申请设备的中断，并在最后一次关闭时释放中断以尽量减少中断被这个设备占用的时间，但是，在大多数情况下为求省事，大多数驱动工程师还是将中断申请和释放的工作放在了设备驱动的模块加载和卸载函数中。

3.软中断

软中断是用软件方式模拟硬件中断的概念，实现宏观上的异步执行效果。事实上， tasklet也是基于软中断实现的。

第 11章异步 I/O所基于的信号也类似于中断。现在，总结一下硬中断、软中断和信号的区别：硬中断是外部设备对 CPU的中断，软中断是程序对内核的中断，而信号则是由内核（或其他进程）对某个进程的中断。

在 Linux内核中，用 softirq_action结构体表征一个软中断，这个结构体中包含软中断处理函数指针和传递给该函数的参数。使用 open_softirq（）函数可以注册软中断对应的处理函数，而 raise_softirq（）函数可以触发一个软中断。

4.三种底半部处理程序所处上下文

软中断和 tasklet仍然运行于中断上下文，而工作队列则运行于进程上下文。因此，在软中断和 tasklet处理函数中不能睡眠，而工作队列处理函数中允许睡眠。因为在中断上下文中睡眠，会导致相应的硬件中断得不到及时的响应。 local_bh_disable（）和 local_bh_enable（）是内核中用于禁止和使能软中断和 tasklet底半部机制的函数。

8.2.4　中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，像 pmic芯片就会报出供电正常、充电完成甚至触摸屏被触摸等多种中断，而这些中断都是通过一根中断线接到 CPU的。 Linux 2.6支持这种中断共享。下面是中断共享的使用方法。

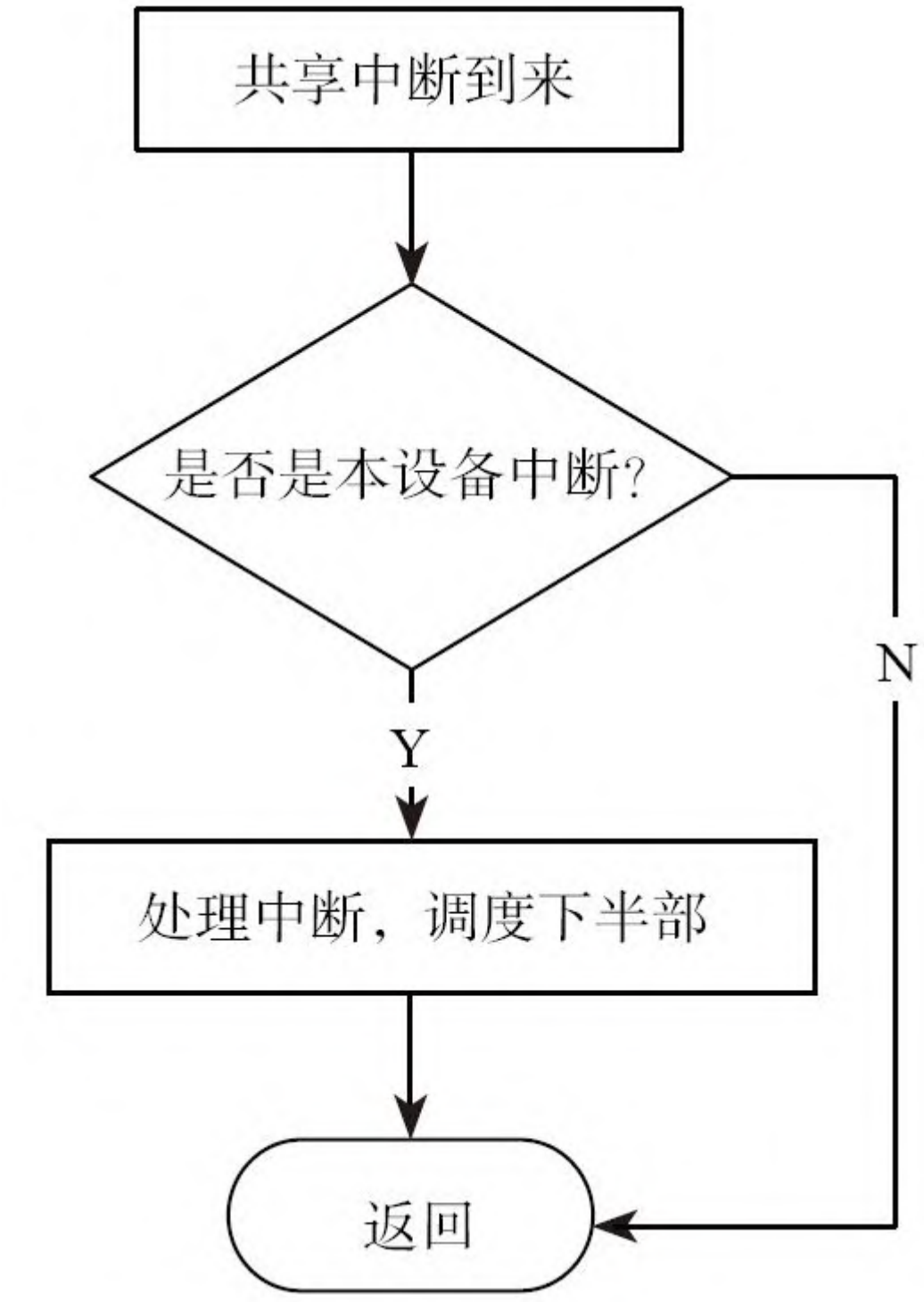


图 8-1　共享中断的处理

• 共享中断的多个设备在申请中断时都应该使用 SA_SHIRQ标志，而且一个设备以 SA_SHIRQ申请某中断成功的前提是之前申请该中断的所有

设备也都以 SA_SHIRQ标志申请该中断。

• 尽管内核模块可访问的全局地址都可以作为 request_irq（）的最后一个参数 dev_id，但是设备结构体指针是可传入的最佳参数。

• 在中断到来时，所有共享此中断的中断处理程序都会被执行，在中断处理程序顶半部中应迅速地根据硬件寄存器中的信息比照传入的dev_id参数判断是否是本设备的中断，若不是应迅速返回，如图 8-1所示。

代码清单 8-3给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

代码清单 8-3　共享中断编程的模板

```
/*中断处理顶半部*/
irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
    ...
    int status = read_int_status();/*获知中断源*/
    if(!is_myint(dev_id,status))/*判断是否是本设备中断
*/
    {
        return IRQ_NONE; /*立即返回
*/
    }
    ...
    return IRQ_HANDLED;
}

/*设备驱动模块加载函数*/
int xxx_init(void)
{
    ...
    /*申请共享中断*/
    result = request_irq(sh_irq, xxx_interrupt, SA_SHIRQ,
"xxx", xxx_dev);
    ...
}
```

```
/*设备驱动模块卸载函数*/
void xxx_exit(void)
{
    ...
    /*释放中断*/
    free_irq(xxx_irq, xxx_interrupt);
    ...
}
```

8.3 Linux定时器

时钟是 CPU运行的脉搏，定时器就是对机器时钟中断的运用，其意义重大。

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后检测各定时器是否到时间，到时间后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序执行update_process_timers（）函数，该函数调用run_local_timers（）函数，这个函数处理 TIMER_SOFTIRQ软中断，运行当前处理器上到期的所有定时器。

在 Linux设备驱动编程中，可以利用 Linux内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动开发工程师多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux内核所提供的用于操作定时器的数据结构和函数如下所示。

1.timer_list

在 Linux内核中， timer_list结构体的一个实例对应一个定时器，如代码清单 8-4所示。

代码清单 8-4 timer_list结构体

```
1 struct timer_list {
2     struct list_head entry; //定时器列表
3     unsigned long expires; //定时器到期时间
4     void (*function)(unsigned long); //定时器处理函数
5     unsigned long data; //作为参数被传入定时器处理函数
6     struct timer_base_s *base;
7 };
```

当定时器期满后，其中第 4行的 function（）成员将被执行，而第 5行的 data成员则是传入其中的参数，第 3行的 expires则是定时器到期的时间（以 jiffy为单位）。

如下代码定义一个名为 my_timer的定时器：

```
struct timer_list my_timer;
```

2.初始化定时器

```
void init_timer（struct timer_list*timer）;
```

上述 init_timer（）函数初始化 timer_list结构体 entry成员的 next为 NULL，并给 base指针赋值。

TIMER_INITIALIZER（ _function， _expires， _data）宏用于赋值定时器结构体的 function、 expires、 data和 base成员，这个宏的定义如下所示：

```
#define TIMER_INITIALIZER(_function, _expires, _data) { \
.function = (_function), \
.expires = (_expires), \
.data = (_data), \
.base = &__init_timer_base, \
}
```

DEFINE_TIMER（ _name， _function， _expires， _data）宏是定义并初始化定时器成员的“快捷方式”，这个宏定义如下所示：

```
#define DEFINE_TIMER(_name, _function, _expires, _data) \
```

```
struct timer_list _name = \
TIMER_INITIALIZER(_function, _expires, _data)
```

此外， setup_timer（）也可用于初始化定时器并赋值其成员，其源代码如下：

```
static inline void setup_timer(struct timer_list * timer,void
(*function)(unsigned long), unsigned long data)
{
    timer->function = function;
    timer->data = data;
    init_timer(timer);
}
```

3.增加定时器

```
void add_timer（struct timer_list*timer）;
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4.删除定时器

```
int del_timer（struct timer_list*timer）;
```

上述函数用于删除定时器。

del_timer_sync（）是 del_timer（）的同步版，主要在多处理器系统中使用，如果编译内核时不支持 SMP， del_timer_sync（）和 del_timer（）等价。

5.修改定时器的 expire

```
int mod_timer（struct timer_list*timer，unsigned long
expires）；
```

上述函数用于修改定时器的到期时间，在新的被传入的 expires到来后才会执行定时器函数。

代码清单 8-5给出了一个完整的内核定时器使用模板，在大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 8-5　内核定时器使用模板

```
1 /*xxx设备结构体*/
2 struct xxx_dev
3 {
4     struct cdev cdev;
5     ...
6     timer_list xxx_timer;/*设备要使用的定时器*/
7 };
8
9 /*xxx驱动中的某函数*/
10 xxx_func1(...)
11 {
12     struct xxx_dev *dev = filp->private_data;
13     ...
14     /*初始化定时器*/
15 init_timer(&dev->xxx_timer);
16 dev->xxx_timer.function = &xxx_do_timer;
17 dev->xxx_timer.data = (unsigned long)dev;
18     /*设备结构体指针作为定时器处理函数参数*/
19     dev->xxx_timer.expires = jiffies + delay;
20 /*添加（注册）定时器*/
21     add_timer(&dev->xxx_timer);
22     ...
23 }
24
25 /*xxx驱动中的某函数*/
26 xxx_func2(...)
27 {
28     ...
```

```
29      /*删除定时器*/
30      del_timer (&dev->xxx_timer);
31      ...
32  }
33
34  /*定时器处理函数*/
35  static void xxx_do_timer(unsigned long arg)
36  {
37  struct xxx_device *dev = (struct xxx_device *)(arg);
38  ...
39  /*调度定时器再执行*/
40  dev->xxx_timer.expires = jiffies + delay;
41  add_timer(&dev->xxx_timer);
42  ...
43  }
```

从代码清单第 19、 40行可以看出，定时器的到期时间往往是在目前jiffies的基础上添加一个时延，若为 HZ，则表示延迟 1s。

在定时器处理函数中，在做完相应的工作后，往往会延后 expires并将定时器再次添加到内核定时器链表，以便定时器能再次被触发。

8.4 Linux延时处理

在 Linux内核开发中，还有一个经常会用到的技术，它是与时间中断紧密联系的，即延时技术。

8.4.1 短延时

Linux内核中提供了如下 3个函数分别进行纳秒、微秒和毫秒的延迟。

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待，它根据 CPU频率进行一定次数的循环。有时候，可以在软件中进行这样的延迟：

```
void delay(unsigned int time)
{
    while (time--);
}
```

ndelay（）、 udelay（）和 mdelay（）函数的实现方式机理与此类似。

毫秒时延（以及更大的秒时延）已经比较长了，在内核中最好不要直接使用 mdelay（）函数，这将无谓地耗费 CPU资源，对于毫秒级以上时延，内核提供了下述函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠函数参数所指定的时间，msleep（）、 ssleep（）所启动的睡眠不能被打断，而msleep_interruptible（）则可以被打断。

注：受系统 HZ以及进程调度的影响， msleep（）类似函数的精度是有限的。对于时间精度要求较高的模块，例如 3G通信模块，在使用定时器时，就要确定好睡眠的时间，参考精度，给予一定的冗余。

8.4.2 长延时

对于延时是秒级以上的，则不宜再使用前面所述的短延时的方式来实现延时等待。

在 Linux内核中，进行延迟的一个很直观的方法是比较当前的jiffies和目标 jiffies（目标 jiffies就是当前 jiffies加上要延迟时间的 jiffies），直到未来的 jiffies达到目标 jiffies。代码清单 8-6给出了使用忙等待先延迟 100个 jiffies再延迟 2s的实例。

代码清单 8-6 忙等待时延实例

```
1 /*延迟100个jiffies*/
2 unsigned long delay = jiffies + 100;
3 while (time_before(jiffies, delay));
4
5 /*再延迟2s*/
6 unsigned long delay = jiffies + 2*HZ;
7 while (time_before(jiffies, delay));
```

与 time_before（）对应的还有一个 time_after（），它们在内核中定义为（实际上只是将传入的未来时间 jiffies和被调用时的jiffies进行一个简单的比较）：

```
#define time_after(a,b) \
(typecheck(unsigned long, a) && \
typecheck(unsigned long, b) && \
```

```
((long)(b) - (long)(a) < 0))
#define time_before(a,b) time_after(b,a)
```

为了防止 time_before（）和 time_after（）的比较过程中编译器对jiffies的优化，内核将其定义为 volatile变量，这样该变量就不会被 CPU高速缓冲，从而保证它每次都被重新读取。

8.4.3 睡眠延时

前两节的延时等待对宝贵的 CPU资源是不利的。尤其是长时间等待更是对该计算资源的严重浪费。因此 Linux内核又引入了睡眠延时技术。

当使用睡眠技术时，随着延迟在等待的时间到来过程中，进程处于睡眠状态， CPU资源可让给其他进程使用。 schedule_timeout（）可以使当前任务睡眠指定的 jiffies之后，重新被调度执行。 msleep（）和 msleep_interruptible（）在本质上都是依靠包含了schedule_timeout（）的 schedule_timeout_uninterruptible（）和schedule_timeout_interruptible（）实现的，如代码清单 8-7所示。

代码清单 8-7　schedule_timeout（）的使用

```
1 void msleep(unsigned int msecs)
2 {
3     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5     while (timeout)
6         timeout = schedule_timeout_uninterruptible(timeout);
7 }
8
9 unsigned long msleep_interruptible(unsigned int msecs)
10 {
11     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13     while (timeout && !signal_pending(current))
14         timeout = schedule_timeout_interruptible(timeout);
15     return jiffies_to_msecs(timeout);
16 }
```

实际上， schedule_timeout（）的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒参数对应的进程。

代码清单 8-7的第 6行和第 14行分别调用 schedule_timeout_uninterruptible（）和 schedule_timeout_interruptible（）。这两个函数的区别在于，前者在调用 schedule_timeout（）之前置进程状态为 TASK_UNINTERRUPTIBLE，后者置进程状态为 TASK_INTERRUPTIBLE，如代码清单 8-8所示。

代码清单 8-8　schedule_timeout_interruptible（）和 schedule_timeout_uninterruptible（）

```
signed long _ _sched schedule_timeout_interruptible(signed long
timeout)
{
    _ _set_current_state(TASK_INTERRUPTIBLE);
    return schedule_timeout(timeout);
}

signed long _ _sched schedule_timeout_uninterruptible(signed
long timeout)
{
    _ _set_current_state(TASK_UNINTERRUPTIBLE);
    return schedule_timeout(timeout);
}
```

另外，下面两个函数可以将当前进程添加到等待队列中，从而在等待队列上睡眠。当超时发生时，进程将被唤醒，如下所示：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned
long timeout);
```

其中 interruptible_sleep_on_timeout（）让进程的睡眠在超时前被打断。关于 TASK_INTERRUPTIBLE与 TASK_UNINTERRUPTIBLE，以及等待队列的具体含义，我们将在后面章节展开讨论。理解它们的概念后，再回过头来看看本节将会有更深刻的体会。

第9章　Linux设备驱动中的并发

通过第7、8章的学习，我们理清了进程与设备进行交互的基本流程。但我们知道，这些设备常不是给一个用户、一个任务或一个进程使用的。

因此，Linux设备驱动中必须解决的一个问题是：多个进程对共享资源的并发访问。而多进程对同一个设备的并发访问，势必会导致对该设备资源的竞争。

9.1　Linux中的并发与竞争

并发（ concurrency）指的是多个执行单元同时、并行被执行，而并发的执行单元对共享资源（硬件资源和软件上的全局变量、静态变量等）的访问则很容易导致竞态（ race conditions）。例如，对于6.2节中所列举的 virtualchar设备，假设一个执行单元 A对其写入3000个字符 "a"，而另一个执行单元 B对其写入 4000个字符 "b"，第三个执行单元 C读取 virtualchar的所有字符。如果执行单元 A、 B的写操作以如图 9-1所示顺序执行，执行单元 C的读操作不会有问题。但是，如果执行单元 A、 B以如图 9-2所示顺序执行，而执行单元 C又“不合时宜”地读，则会读出 3000个 "b"。

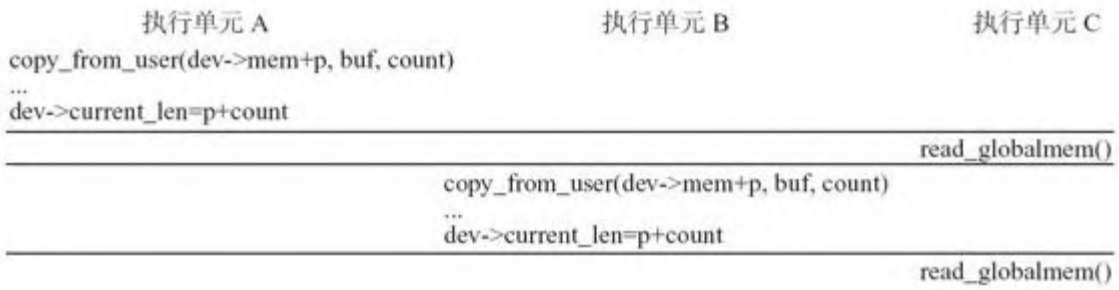


图9-1　并发执行单元的顺序执行

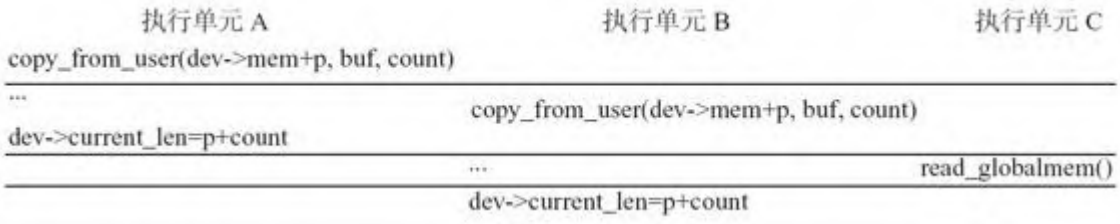


图9-2　并发执行单元的交错执行

比图 9-2更复杂、更混乱的并发大量地存在于设备驱动中，只要并发的多个执行单元存在对共享资源的访问，竞态就可能发生。在 Linux内核中，主要的竞态发生于如下几种情况。

1.对称多处理器（SMP）的多个CPU

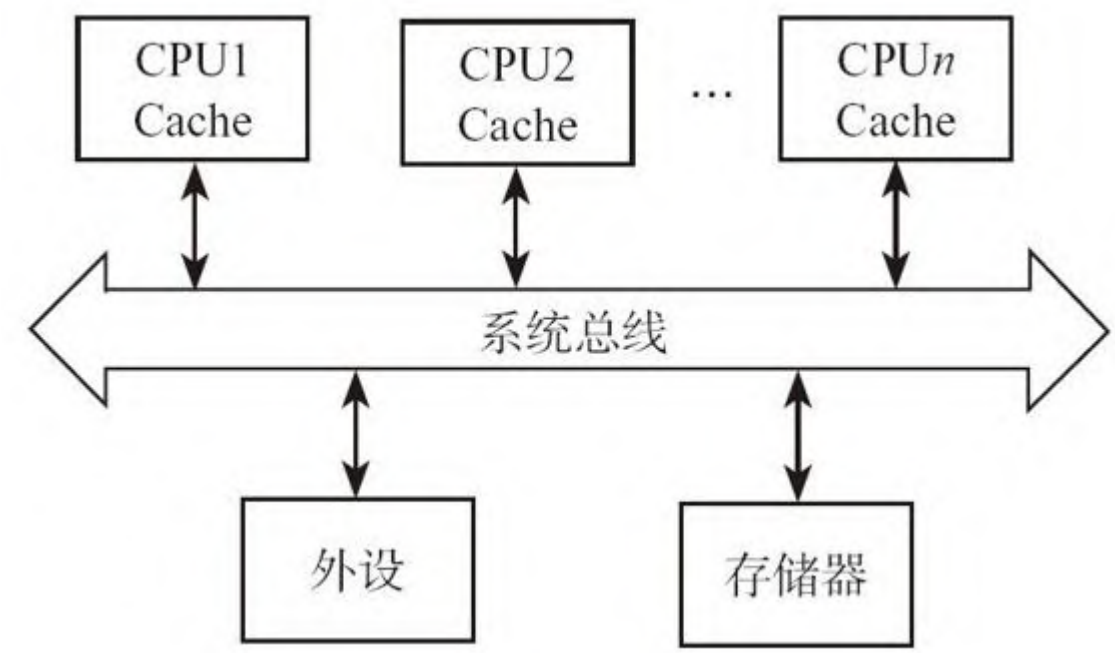


图9-3　SMP体系结构

SMP是一种紧耦合、共享存储的系统模型，其体系结构如图 9-3所示，它的特点是多个 CPU使用共同的系统总线，因此可访问共同的外设和存储器。

2.单CPU内进程与抢占它的进程

Linux 2.6内核支持抢占调度，一个进程在内核执行的时候可能被另一高优先级进程打断，进程与抢占它的进程访问共享资源的情况类似于SMP的多个 CPU。

3.中断（硬中断、软中断、tasklet、底半部）与进程之间

中断可以打断正在执行的进程，如第 8章所讲，中断处理程序是独立于进程的执行单元。如果中断处理程序访问进程正在访问的资源，则竞态也会发生。

此外，中断也有可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态。

上述并发的发生情况除了 SMP是真正的并行以外，其他都是“宏观并行、微观串行”的，但其引发的实质问题与 SMP相似。

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他执行单元被禁止访问。

访问共享资源的代码区域称为临界区（ critical section），临界区需要以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁和信号量等是 Linux设备驱动中可采用的互斥途径。在接下来的章节里，我们将对这些互斥机制进行一一讲解。

9.2　Linux中常用的同步访问技术

9.2.1　中断屏蔽

在单 CPU范围内避免竞态的一种简单方法是：在进入临界区之前屏蔽系统的中断。 CPU一般都具备屏蔽中断和打开中断的功能。很容易理解，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生。而且，由于 Linux内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也就得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable() //屏蔽中断
...
critical section //临界区
...
local_irq_enable() //开中断
```

由于 Linux系统的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。这就要求在屏蔽中断后，当前的内核执行路径应当尽快地执行完临界区的代码。

local_irq_disable（）和 local_irq_enable（）都只能禁止和使能本 CPU内的中断，因此，并不能解决 SMP多 CPU引发的竞态。因此，单独使用中断屏蔽通常不是一种值得推荐的避免竞态的方法，它适宜与自旋锁联合使用。

与 local_irq_disable（）不同的是， local_irq_save（ flags）除了进行禁止中断的操作以外，还保存目前 CPU的中断位信息，local_irq_restore（ flags）完成的是与 local_irq_save（ flags）相反的操作。

如果只是想禁止中断的底半部，应使用 local_bh_disable（），使能被 local_bh_disable（）禁止的底半部应该调用 local_bh_enable（）。

9.2.2 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

Linux内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不会被打断。位和整型变量原子操作都依赖底层 CPU的原子操作来实现，因此所有这些函数都与 CPU架构密切相关。

1.整型原子操作

（ 1）设置原子变量的值

```
void atomic_set(atomic_t *v, int i); //设置原子变量的值为i
atomic_t v = ATOMIC_INIT(0); //定义原子变量v并初始化为0
```

（ 2）获取原子变量的值

```
atomic_read（atomic_t*v）; //返回原子变量的值
```

（ 3）原子变量加 /减

```
void atomic_add(int i, atomic_t *v); //原子变量增加i
void atomic_sub(int i, atomic_t *v); //原子变量减少i
```

（ 4）原子变量自增 /自减

```
void atomic_inc(atomic_t *v); //原子变量增加1
void atomic_dec(atomic_t *v); //原子变量减少1
```

（ 5）操作并测试

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

上述操作对原子变量执行自增、自减和减操作后（注意没有加）测试其是否为 0，为 0则返回 true，否则返回 false。

（ 6）操作并返回

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

上述操作对原子变量进行加 /减和自增 /自减操作，并返回新的值。

2.位原子操作

（ 1）设置位

```
void set_bit（nr, void*addr）;
```

上述操作设置 addr地址的第 nr位，所谓设置位即将位写为 1。

（ 2）清除位

```
void clear_bit（nr，void*addr）；
```

上述操作清除 addr地址的第 nr位，所谓清除位即将位写为 0。

（ 3）改变位

```
void change_bit（nr，void*addr）；
```

上述操作对 addr地址的第 nr位进行反置。

（ 4）测试位

```
test_bit（nr，void*addr）；
```

上述操作返回 addr地址的第 nr位。

（ 5）测试并操作位

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

上述 test_and_xxx_bit（ nr， void*addr）操作等同于执行 test_bit（ nr， void*addr）后再执行 xxx_bit（ nr， void*addr）。

代码清单 9-1给出了原子变量的使用实例，它用于使设备最多只能被一个进程打开。

代码清单 9-1　使用原子变量使设备只能被一个进程打开

```
static atomic_t xxx_available = ATOMIC_INIT(1); /*定义原子变量*/

static int xxx_open(struct inode *inode, struct file *filp)
{
    ...
    if (!atomic_dec_and_test(&xxx_available))
    {
        atomic_inc(&xxx_available);
        return - EBUSY; /*已经打开*/
    }
    ...
    return 0; /* 成功*/
}

static int xxx_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&xxx_available); /* 释放设备*/
    return 0;
}
```

9.2.3 自旋锁

由于中断屏蔽会使得中断得不到响应，从而导致系统性能变差；而原子操作受限于 CPU，只能实现有限几种基本数据类型的排他操作。为此， Linux内核又设计了自旋锁以实现共享资源的同步访问。

1.自旋锁的使用

自旋锁（ spin lock）是一种对临界资源进行互斥访问的典型手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU上运行的代码需先执行一个原子操作，该操作测试并设置（ test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。

如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，说白了就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。

理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区或者标记为“我当前在运行，请稍等一会”，或者标记为“我当前不在运行，可以被使用”。如果 A执行单元首先进入例程，它将持有自旋锁；当 B执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A执行单元释放后才能进入。

Linux系统中与自旋锁相关的操作主要有如下 4种。

（ 1）定义自旋锁

```
spinlock_t spin;
```

（ 2）初始化自旋锁

```
spin_lock_init（lock）
```

该函数用于动态初始化自旋锁 lock。

（ 3）获得自旋锁

```
spin_lock（lock）
```

以上函数用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放。

```
spin_trylock（lock）
```

以上函数尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，而不再“在原地打转”。

（ 4）释放自旋锁

```
spin_unlock（lock）
```

以上函数释放自旋锁 lock，它与 spin_trylock或 spin_lock配对使用。自旋锁一般这样被使用，如下所示：

```
//定义一个自旋锁
spinlock_t lock;
spin_lock_init(&lock);
spin_lock (&lock) ; //获取自旋锁,保护临界区
...//临界区
spin_unlock (&lock) ; //解锁
```

自旋锁主要针对 SMP或单 CPU但内核可抢占的情况，对于单 CPU且内核不支持抢占的系统，自旋锁退化为空操作。在单 CPU且内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU系统的行为实际类似于 SMP系统，因此，在这样的单 CPU系统中使用自旋锁仍十分必要。

很容易理解，中断处理程序中不能使用自旋锁，因为中断处理程序中申请的自旋锁被其他执行路径所占有，会导致系统其他中断得不到响应。所以尽管用了自旋锁可以保证临界区不受别的 CPU和本 CPU内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候，还可能受到中断和底半部（ BH）的影响。为了防止这种影响，就需要用到自旋锁的衍生。 spin_lock（） /spin_unlock（）是自旋锁机制的基础，它们与关中断 local_irq_disable（） /开中断 local_irq_enable（）、关底半部 local_bh_disable（） /开底半部 local_bh_enable（）、关中断并保存状态字 local_irq_save（） /开中断并恢复状态 local_irq_restore（）结合就形成了整套自旋锁机制，关系如下所示：

```
spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_unlock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()
```

这样，就可以控制进程与中断处理程序这两个独立执行单元对临界区的同步访问。这就保证了已拥有自旋锁的进程，能够在本 CPU下顺利地完成相应临界区的执行，而不致阻塞死锁。

驱动开发工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题。

1）自旋锁实际上是忙等锁，当锁不可用时， CPU一直循环执行“测试并设置”该锁直到可用而取得该锁， CPU在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。

2）自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU想第二次获得这个自旋锁，则该 CPU将产生死锁。此外，如果进程获得自旋锁之后再阻塞，也有可能导致死锁的发生。 copy_from_user（）、copy_to_user（）和 kmalloc（）等函数都可能由于内存缺页而引起阻塞，而如果拥有这些内存的执行路径又被阻塞在这个自旋锁上，结果就是死锁，因此在自旋锁的占用期间不能调用这些函数。

代码清单 9-2给出了自旋锁的使用实例，它被用于实现设备只能被至多一个进程打开。

代码清单 9-2　使用自旋锁使设备只能被一个进程打开

```
int xxx_count = 0;/*定义文件打开次数计数*/

static int xxx_open(struct inode *inode, struct file *filp)
{
```

```
    ...
    spinlock(&xxx_lock);
    if (xxx_count)/*已经打开*/
    {
        spin_unlock(&xxx_lock);
        return - EBUSY;
    }
    xxx_count++;/*增加使用计数*/
    spin_unlock(&xxx_lock);
    ...
    return 0; /* 成功*/
}

static int xxx_release(struct inode *inode, struct file *filp)
{
    ...
    spinlock(&xxx_lock);
    xxx_count--; /*减少使用计数*/
    spin_unlock(&xxx_lock);

    return 0;
}
```

2.读写自旋锁

自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生读写自旋锁（ rwlock）可允许读的并发。

读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面只能最多有一个写进程，而在读操作方面可以同时有多个读执行单元。当然，读和写也不能同时进行。

读写自旋锁涉及的操作如下所示。

（ 1）定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态初始化*/
```

```
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化*/
```

（ 2）读锁定

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

（ 3）读解锁

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。

read_lock_irqsave（）、 read_lock_irq（）和 read_lock_bh（）分别是 read_lock（）与 local_irq_save（）、 local_irq_disable（）和 local_bh_disable（）的组合，读解锁函数 read_unlock_irqrestore（）、 read_unlock_irq（）、 read_unlock_bh（）的情况与此类似。

（ 4）写锁定

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

（ 5）写解锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long
flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

write_lock_irqsave（）、 write_lock_irq（）、 write_lock_bh（）分别是 write_lock（）与 local_irq_save（）、 local_irq_disable（）和 local_bh_disable（）的组合，写解锁函数 write_unlock_irqrestore（）、 write_unlock_irq（）、 write_unlock_bh（）的情况与此类似。

在对共享资源进行写之前，应该先调用写锁定函数，完成之后应调用写解锁函数。与 spin_trylock（）一样， write_trylock（）也只是尝试获取读写自旋锁，不管成功失败都会立即返回。

读写自旋锁一般这样被使用，如下所示：

```
rwlock_t lock; //定义rwlock
rwlock_init(&lock); //初始化rwlock
//读时获取锁
read_lock(&lock);
... //临界资源
read_unlock(&lock);
//写时获取锁
write_lock_irqsave(&lock, flags);
... //临界资源
write_unlock_irqrestore(&lock, flags);
```

3.顺序锁

顺序锁（ seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元可以在写执行

单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。

但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须“自旋”在那里，直到写执行单元释放了顺序锁。

如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。在读写同时进行的概率比较小时，这种锁的性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

顺序锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致内核异常（ Oops）。

在 Linux内核中，写执行单元涉及如下顺序锁操作。

（ 1）获得顺序锁

```
void write_seqlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
write_seqlock_irqsave(lock, flags)
write_seqlock_irq(lock)
write_seqlock_bh(lock)
```

其中：

```
write_seqlock_irqsave() = loal_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

（ 2）释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中：

```
write_sequnlock_irqrestore() = write_sequnlock() +
local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
...//写操作代码块
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与 spinlock相同。

读执行单元涉及如下顺序锁操作。

（ 1）读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```

读执行单元在对被顺序锁 s1保护的共享资源进行访问前需要调用该函数，该函数仅返回顺序锁 s1的当前顺序号。其中：

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

（ 2）重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 s1保护的共享资源后需要调用该函数，来检查在读访问期间是否有写操作。如果有写操作，读执行单元就需要重新进行读操作。其中：

```
read_seqretry_irqrestore() = read_seqretry() +
local_irq_restore()
```

读执行单元使用顺序锁的模式如下：

```
do {
    seqnum = read_seqbegin(&seqlock_a);
    //读操作代码块
    ...
} while (read_seqretry(&seqlock_a, seqnum));
```

9.2.4 信号量

信号量（ semaphore）是用于保护临界区的另一种常用方法，它的使用方式与自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。

1.信号量的使用

Linux系统中与信号量相关的操作主要有如下 4种。

（ 1）定义信号量

下列代码定义名称为 sem的信号量。

```
struct semaphore sem;
```

（ 2）初始化信号量

```
void sema_init（struct semaphore*sem，int val）；
```

以上函数初始化信号量，并设置信号量 sem的值为 val。尽管信号量可以被初始化为大于 1的值从而成为一个计数信号量，但是它通常不被这样使用。

```
void init_MUTEX（struct semaphore*sem）；
```

以上函数用于初始化一个用于互斥的信号量，它把信号量 sem的值设置为 1，等同于 sema_init（ struct semaphore*sem， 1）。

```
void init_MUTEX_LOCKED（struct semaphore*sem）；
```

以上函数也用于初始化一个信号量，但它把信号量 sem的值设置为 0，等同于 sema_init（ struct semaphore*sem， 0）。

此外，下面两个宏是定义并初始化信号量的“快捷方式”。

```
DECLARE_MUTEX(name)
DECLARE_MUTEX_LOCKED(name)
```

前者定义一个名为 name的信号量并初始化为 1，后者定义一个名为 name的信号量并初始化为 0。

（ 3）获得信号量

```
void down（struct semaphore*sem）;
```

以上函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文使用。

```
int down_interruptible（struct semaphore*sem）;
```

以上函数功能与 down（）类似，不同之处为，因为 down（）而进入睡眠状态的进程不能被信号打断，而因为 down_interruptible（）进入睡眠状态的进程能被信号打断，信号也会导致该函数返回，以响应系统的某些状态变化。这时候该函数的返回值非 0。

```
int down_trylock（struct semaphore*sem）;
```

以上函数尝试获得信号量 sem，如果能够立刻获得，它就返回 0，否则返回非 0值。它不会导致调用者睡眠，可以在中断上下文使用。

在使用 down_interruptible（）获取信号量时，一般对返回值进行检查，如果非 0，通常立即返回 -ERESTARTSYS，如：

```
if (down_interruptible(&sem))
```

```
{
    return - ERESTARTSYS;
}
```

（ 4）释放信号量

```
void up（struct semaphore*sem）;
```

以上函数释放信号量 sem，唤醒等待者。

信号量一般这样被使用，如下所示：

```
//定义信号量
DECLARE_MUTEX(mount_sem);
down(&mount_sem);//获取信号量,保护临界区
...
critical section //临界区
...
up(&mount_sem);//释放信号量
```

注： Linux自旋锁和信号量所采用的“获取锁—访问临界区—释放锁”的方式存在于几乎所有的多任务操作系统之中。

代码清单 9-3给出了使用信号量实现设备只能被一个进程打开的例子，等同于代码清单 9-1和代码清单 9-2。

代码清单 9-3 使用信号量实现设备只能被一个进程打开

```
static DECLARE_MUTEX(xxx_lock);//定义互斥锁

static int xxx_open(struct inode *inode, struct file *filp)
{
    ...
    if (down_trylock(&xxx_lock)) //获得打开锁
```

```
        return - EBUSY; //设备忙
    ...
    return 0; /* 成功*/
}

static int xxx_release(struct inode *inode, struct file *filp)
{
    up(&xxx_lock); //释放打开锁
    return 0;
}
```

2.信号量用于同步

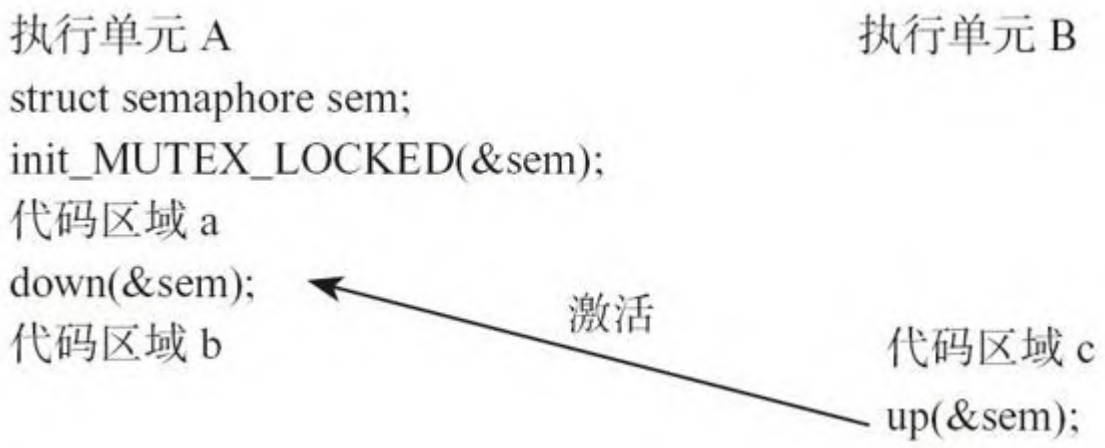


图 9-4　信号量用于同步

信号量除了可以用来保护临界资源，还可用于两个执行单元间的同步通信。同步意味着一个执行单元的继续执行需等待另一执行单元完成某事，保证执行的先后顺序。如果信号量被初始化为 0，则它可以用于同步。如图 9-4所示，执行单元 A执行代码区域 b之前，必须等待执行单元 B执行完代码区域 c，信号量可辅助这一同步过程。

3.完成信号量用于同步

Linux系统提供了一种比信号量更好的同步机制，即完成信号量（completion，关于这个名词，至今没有好的翻译，笔者将其译为“完成信号量”），它用于一个执行单元等待另一个执行单元完成某操作。

Linux系统中与 completion相关的操作主要有以下 4种。

（ 1）定义完成信号量

下列函数定义名为 my_completion的完成信号量。

```
struct completion my_completion;
```

（ 2）初始化完成信号量

下列函数初始化 my_completion这个完成信号量。

```
init_completion（&my_completion）;
```

对 my_completion的定义和初始化可以通过如下快捷方式实现。

```
DECLARE_COMPLETION（my_completion）;
```

（ 3）等待完成信号量

下列函数用于等待一个 completion被唤醒。

```
void wait_for_completion（struct completion*c）;
```

（ 4）唤醒完成信号量

下面两个函数用于唤醒完成信号量。

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

前者只唤醒一个等待的执行单元，后者释放所有等待同一完成信号量的执行单元。图 9-5描述了使用完成信号量实现的同步功能。

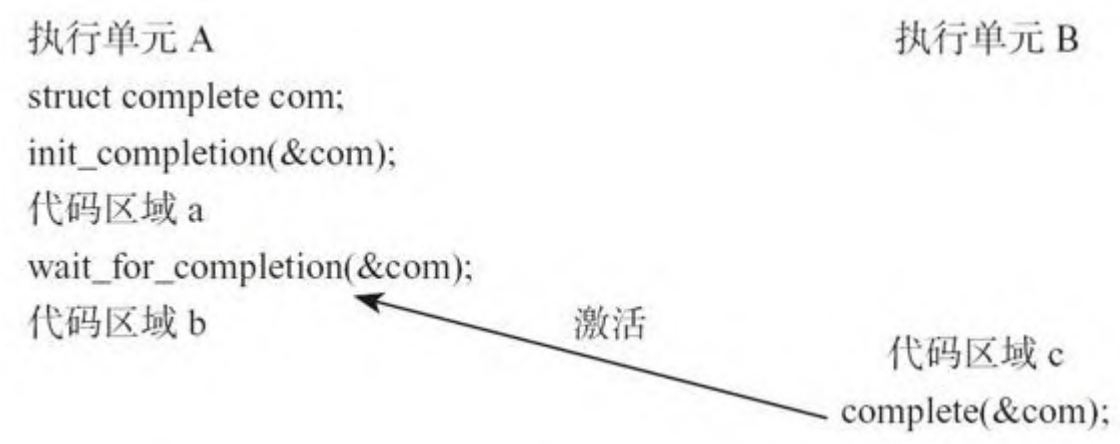


图 9-5　完成信号量用于同步

4.自旋锁与信号量的选用

自旋锁和信号量都是解决互斥操作的基本手段，面对特定的情况，应该如何进行选择呢？选择的依据是临界区的性质和系统的特点。

从严格意义上说，信号量和自旋锁属于不同层次的互斥手段，前者的实现依赖于后者。在信号量本身的实现上，为了保证信号量结构存取的原子性，在多 CPU中需要自旋锁来互斥。

信号量是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程为单位，代表进程来争夺资源的。如果竞争失败，会发生进程上下文切换，当前进程进入睡眠状态， CPU将运行其他进程。鉴于进程上下文切换的开销也很大，因此，只有当进程占用资源时间较长时，用信号量才是较好的选择。

当所要保护的临界区访问时间比较短时，用自旋锁是非常方便的，因为它节省上下文切换的时间。但是 CPU得不到自旋锁会在那里空转直

到其他执行单元解锁为止，所以要求锁不能在临界区长时间停留，否则会降低系统的效率。

由此，可以总结出自旋锁和信号量选用的 3项原则。

1）当锁不能被获取时，使用信号量的开销是进程上下文切换时间Tsw，使用自旋锁的开销是等待获取自旋锁（由临界区执行时间决定）Tcs，若 Tcs比较小，应使用自旋锁；若 Tcs很大，应使用信号量。

2）信号量所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就可能发生。

3）信号量存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在信号量和自旋锁之间只能选择自旋锁。当然，如果一定要使用信号量，则只能通过 down_trylock（）方式进行，不能获取就立即返回且不执行临界区代码，以避免阻塞。

5.读写信号量

读写信号量与信号量的关系与读写自旋锁和自旋锁的关系类似，读写信号量可能引起进程阻塞，但它可允许 N个读执行单元同时访问共享资源，而最多只能有一个写执行单元。因此，读写信号量是一种相对放宽条件的粒度稍大于信号量的互斥机制。

读写信号量涉及的操作包括如下 5种。

（ 1）定义和初始化读写信号量

```
struct rw_semaphore my_rws; /*定义读写信号量*/
void init_rwsem(struct rw_semaphore *sem); /*初始化读写信号量*/
```

（ 2）读信号量获取

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
```

（ 3）读信号量释放

```
void up_read（struct rw_semaphore*sem）;
```

（ 4）写信号量获取

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
```

（ 5）写信号量释放

```
void up_write（struct rw_semaphore*sem）;
```

读写信号量一般这样被使用，如下所示：

```
rw_semaphore rw_sem; //定义读写信号量
init_rwsem(&rw_sem); //初始化读写信号量
//读时获取信号量
down_read(&rw_sem);
... //临界资源
up_read(&rw_sem);
//写时获取信号量
down_write(&rw_sem);
... //临界资源
up_write(&rw_sem);
```

9.2.5 互斥灯

尽管信号量已经可以实现互斥的功能，而且包含DECLARE_MUTEX（）、 init_MUTEX（）等定义信号量的宏或函数，从名字上看就体现出了“互斥灯”的概念，但是互斥灯（ mutex）在Linux内核中还是真实地存在的。作为工程师来讲，不妨把互斥灯当作特殊的信号量：也就是互斥灯严格用于保护临界区，不应该用来实现两执行单元间的同步通信。

如下代码定义并初始化名为 example_mutex的互斥灯。

```
struct mutex example_mutex;
mutex_init(&example_mutex);
```

下面的两个函数用于获取互斥灯。

```
void fastcall mutex_lock(struct mutex *lock);
int fastcall mutex_lock_interruptible(struct mutex *lock);
int fastcall mutex_trylock(struct mutex *lock);
```

mutex_lock（）与 mutex_lock_interruptible（）的区别和down（）与 down_interruptible（）的区别完全一致，前者引起的睡眠不能被信号打断，而后者可以。 mutex_trylock（）用于尝试获得mutex，获取不到 mutex时不会引起进程睡眠。

下列函数用于释放互斥灯。

```
void fastcall mutex_unlock（struct mutex*lock）;
```

mutex的使用方法和信号量用于互斥的场合完全一样，如下所示：

```
struct mutex example_mutex; //定义mutex
mutex_init(&example_mutex); //初始化mutex
mutex_lock(&example_mutex); //获取mutex
...//临界资源
mutex_unlock(&example_mutex); //释放mutex
```

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

9.3 增加并发控制的 virtualchar驱动

在 virtualchar的读写函数中，由于要调用 copy_from_user（）、copy_to_user（）这些可能导致阻塞的函数，因此不能使用自旋锁，宜使用信号量。

习惯上，我们愿意把某设备所使用的自旋锁、信号量等辅助手段也放在设备结构中，因此，可如代码清单 9-4那样修改 virtualchar_dev结构体的定义，并在模块初始化函数中初始化这个信号量，如代码清单 9-5所示。

代码清单 9-4 增加并发控制后的 virtualchar设备结构体

```
1 struct virtualchar_dev
2 {
3     struct cdev cdev; /*cdev结构体*/
4     unsigned char mem[MEM_SIZE]; /*全局内存*/
5     struct semaphore sem; /*并发控制用的信号量*/
6 };
```

代码清单 9-5 增加并发控制后的 virtualchar设备驱动模块加载函数

```
1 int virtualchar_init(void)
2 {
3     int result;
4     dev_t devno = MKDEV(virtualchar_major, 0);
5
6     /* 申请设备号*/
7     if (virtualchar_major)
8         result = register_chrdev_region(devno, 1,
"virtualchar");
9     else /* 动态申请设备号*/
10     {
11         result = alloc_chrdev_region(&devno, 0, 1,
"virtualchar");
```

```
12          virtualchar_major = MAJOR(devno);
13      }
14      if (result < 0)
15          return result;
16
17      /* 动态申请设备结构体的内存*/
18      virtualchar_devp = kmalloc(sizeof(struct
virtualchar_dev),GFP_KERNEL);
19      if (!virtualchar_devp) /*申请失败*/
20      {
21          result = - ENOMEM;
22          goto fail_malloc;
23      }
24      memset(virtualchar_devp, 0, sizeof(struct
virutalchar_dev));
25
26      virtualchar_setup_cdev(virtualchar_devp, 0);
27      init_MUTEX(&virtualchar_devp->sem); /*初始化信号量*/
28      return 0;
29
30 fail_malloc: unregister_chrdev_region(devno, 1);
31      return result;
32 }
```

在访问 virtualchar_dev中的共享资源时，须先获取它的信号量，访问完成后随即释放这个信号量。驱动中新的关于内存读、写操作如代码清单 9-6所示。

代码清单 9-6　增加并发控制后的 virtualchar读写操作

```
1 /*增加并发控制后的Virtualchar读函数*/
2 static ssize_t virtualchar_read(struct file *filp, char _
_user *buf,size_t size,
3 loff_t *ppos)
4 {
5      unsigned long p = *ppos;
6      unsigned int count = size;
7      int ret = 0;
8      struct virtualchar_dev *dev = filp->private_data; /*获得设
备结构体指针*/
9
10     /*分析和获取有效的读长度*/
```

```
11      if (p >= MEM_SIZE)
12          return count ? - ENXIO: 0;
13      if (count > MEM_SIZE - p)
14          count = MEM_SIZE - p;
15
16      if (down_interruptible(&dev->sem)) //获得信号量
17      {
18          return - ERESTARTSYS;
19      }
20
21      /*内核空间->用户空间*/
22      if (copy_to_user(buf, (void*)(dev->mem + p), count))
23      {
24          ret = - EFAULT;
25      }
26      else
27      {
28          *ppos += count;
29          ret = count;
30
31          printk(KERN_INFO "read %d bytes(s) from %d\n",
count, p);
32      }
33      up(&dev->sem); //释放信号量
34
35      return ret;
36 }
37
38 /*增加并发控制后的Virtualchar写函数*/
39 static ssize_t virtualchar_write(struct file *filp, const
char _ _user *buf,
40 size_t size, loff_t *ppos)
41 {
42      unsigned long p = *ppos;
43      unsigned int count = size;
44      int ret = 0;
45      struct virtualchar_dev *dev = filp->private_data; /*获得
设备结构体指针*/
46
47      /*分析和获取有效的写长度*/
48      if (p >= MEM_SIZE)
49 return count ? - ENXIO: 0;
50 if (count > MEM_SIZE - p)
51 count = MEM_SIZE - p;
52
53 if (down_interruptible(&dev->sem)) //获得信号量
54 {
```

```
55 return - ERESTARTSYS;
56 }
57 /*用户空间->内核空间*/
58 if (copy_from_user(dev->mem + p, buf, count))
59 ret = - EFAULT;
60 else
61 {
62 *ppos += count;
63 ret = count;
64
65 printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
66 }
67 up(&dev->sem); //释放信号量
68 return ret;
69 }
```

代码第 16～ 19和第 53～ 56行用于获取信号量，如果down_interruptible（）返回值非 0，则意味着其在获得信号量之前已被打断，这时写函数返回 -ERESTARTSYS。代码第 33和第 67行用于在对临界资源访问结束后释放信号量。

除了 virtualchar的读写操作之外，如果在读写的同时，另有执行单元响应 MEM_CLEAR IOCTL命令，也会导致全局内存的混乱，因此，virtualchar_ioctl（）函数也须被重写，如代码清单 9-7所示。

代码清单 9-7　增加并发控制后的 virtualchar设备驱动 ioctl（）函数

```
1 static int virtualchar_ioctl(struct inode *inodep, struct
file *filp,unsigned
2 int cmd, unsigned long arg)
3 {
4 struct virtualchar_dev *dev = filp->private_data; /*获得设备结
构体指针*/
5
6 switch (cmd)
7 {
8 case MEM_CLEAR:
9 if (down_interruptible(&dev->sem))//获得信号量

10 {
```

```
11 return - ERESTARTSYS;
12 }
13 memset(dev->mem, 0, MEM_SIZE);
14 up(&dev->sem); //释放信号量

15
16 printk(KERN_INFO "Virtualchar memory is set to zero\n");
17 break;
18
19 default:
20 return - EINVAL;
21 }
22 return 0;
23 }
```

再次建议我们读者须耐心地读完上面的例子代码，它们并不复杂，而且可以让我们对并发控制访问有一个更感性的认识。

第10章　Linux设备的阻塞式与非阻塞式访问

在第9章，我们已经学习了设备驱动的并发同步访问控制，我们将在本章开始学习上层应用访问Linux设备的相关机制。首先，Linux驱动为上层用户空间访问设备提供了阻塞和非阻塞式两种不同设备访问模式。

10.1　阻塞式与非阻塞式访问

阻塞操作是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。而非阻塞操作的进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

Linux驱动程序通常需要提供这样的能力：当应用程序进行read（）、 write（）等系统调用时，若设备的资源不能获取，而用户又希望以阻塞的方式访问设备，驱动程序应在设备驱动的xxx_read（）、 xxx_write（）等操作中将进程阻塞，直到资源可以获取，此后，应用程序的 read（）、 write（）等调用才返回，整个过程仍然进行了正确的设备访问，而用户并没有感知到；若用户以非阻塞的方式访问设备文件，则当设备资源不可获取时，设备驱动的xxx_read（）、 xxx_write（）等操作应立即返回， read（）、write（）等系统调用也随即被返回。

阻塞从字面上听起来似乎意味着低效率，实则不然，如果设备驱动不阻塞，则用户想获取设备资源只能不停地查询，这反而会无谓地耗费CPU资源。而阻塞访问时，不能获取资源的进程将进入休眠，并将 CPU资源让给其他进程。

因为阻塞的进程会进入休眠状态，因此，必须确保有一个地方能够唤醒休眠的进程。唤醒进程的地方最可能发生在中断，因为硬件资源状态变化往往伴随着一个中断。

代码清单 10-1和代码清单 10-2分别演示了以阻塞和非阻塞方式读取串口一个字符的代码。实际的串口编程中，若使用非阻塞模式，还可借助信号响应（ sigaction）以异步方式访问串口以提高 CPU利用率。而这里仅仅是为了说明阻塞与非阻塞的区别。

代码清单 10-1　阻塞地读取串口一个字符

```
char buf;
```

```
fd = open("/dev/ttyS1", O_RDWR);
...
res = read(fd,&buf,1); //当串口上有输入时才返回
if(res==1)
    printf("%c\n", buf);
```

代码清单 10-2　非阻塞地读取串口一个字符

```
char buf;
fd = open("/dev/ttyS1", O_RDWR| O_NONBLOCK);
...
while(read(fd,&buf,1)!=1); //串口上无输入也返回,所以要循环尝试读取串口
    printf("%c\n", buf);
```

10.1.1　等待队列

在 Linux驱动程序中，可以使用等待队列（ wait queue）来实现阻塞进程的唤醒。 wait queue很早就作为一个基本的功能单位出现在Linux内核里，它以队列为基础数据结构，与进程调度机制紧密结合，能够用于实现内核中的异步事件通知机制。等待队列可以用来同步对系统资源的访问。事实上，第 9章所讲述的信号量在内核中也依赖等待队列来实现。

Linux 2.6提供如下关于等待队列的操作。

1.定义“等待队列头”

```
wait_queue_head_t example_queue;
```

2.初始化“等待队列头”

```
init_waitqueue_head（&example_queue）;
```

而下面的 DECLARE_WAIT_QUEUE_HEAD（）宏可以作为定义并初始化等待队列头的“快捷方式”。

```
DECLARE_WAIT_QUEUE_HEAD（name）
```

3.定义等待队列

```
DECLARE_WAITQUEUE（example，tsk）
```

该宏用于定义并初始化一个名为 example的等待队列。

4.添加 /移除等待队列

```
void fastcall add_wait_queue(wait_queue_head_t *q, wait_queue_t
*wait);
void fastcall remove_wait_queue(wait_queue_head_t *q,
wait_queue_t *wait);
```

add_wait_queue（）用于将等待队列 wait添加到等待队列头 q指向的等待队列链表中，而 remove_wait_queue（）用于将等待队列 wait从附属的等待队列头 q指向的等待队列链表中移除。

5.等待事件

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

等待事件第一个参数 queue作为等待队列头的等待队列被唤醒，而且第二个参数 condition必须满足，否则阻塞。 wait_event（）和wait_event_interruptible（）的区别在于后者可以被信号打断，而前者不能。加上 _timeout后的宏意味着阻塞等待的超时时间，以jiffy为单位，在第三个参数的 timeout到达时，不论 condition是否满足均返回。 wait-event（）的定义如代码清单 10-3所示，从其源代码可以看出，当 condition满足时， wait_event（）会立即返回，否则，阻塞等待 condition满足。

代码清单 10-3　wait_event（）函数

```
1 #define wait_event(wq, condition) \
2 do { \
3     if (condition) /*条件满足立即返回*/ \
4         break; \
5     __wait_event(wq, condition);/*添加等待队列并阻塞*/
6 } while (0)
7
8 #define __wait_event(wq, condition) \
9 do { \
10     DEFINE_WAIT(__wait); \
11     \
12     for (;;) { \
13         prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
\
14         if (condition) \
15             break; \
16         schedule();/*放弃CPU*/ \
17     } \
18     finish_wait(&wq, &__wait); \
19 } while (0)
20
21 void fastcall
22 prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait,
int state)
23 {
24     unsigned long flags;
25
26     wait->flags &= ~WQ_FLAG_EXCLUSIVE;
27     spin_lock_irqsave(&q->lock, flags);
28     if (list_empty(&wait->task_list))    // 判断等待队列的任务
链表为空
```

```
29          __add_wait_queue(q, wait);//添加等待队列
30      if (is_sync_wait(wait))
31          set_current_state(state);//改变当前进程的状态为休眠
32      spin_unlock_irqrestore(&q->lock, flags);
33 }
34
35 void fastcall finish_wait(wait_queue_head_t *q, wait_queue_t
*wait)
36 {
37      unsigned long flags;
38
39      __set_current_state(TASK_RUNNING);// 恢复当前进程的状态为
TASK_RUNNING
40   if (!list_empty_careful(&wait->task_list)) {
41          spin_lock_irqsave(&q->lock, flags);
42          list_del_init(&wait->task_list);
43          spin_unlock_irqrestore(&q->lock, flags);
44      }
45 }
```

6.唤醒队列

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

上述操作会唤醒以 queue作为等待队列头的、所有等待队列中所有属于该等待队列头的等待队列对应的进程。 wake_up（）与 wait_event（）或 wait_event_timeout（）成对使用，而 wake_up_interruptible（）则应与 wait_event_interruptible（）或 wait_event_interruptible_timeout（）成对使用。 wake_up（）可唤醒处于 TASK_INTERRUPTIBLE和 TASK_UNINTERRUPTIBLE的进程，而 wake_up_interruptible（）只能唤醒处于 TASK_INTERRUPTIBLE的进程。

7.在等待队列上睡眠

```
sleep_on(wait_queue_head_t *q );
```

```
interruptible_sleep_on(wait_queue_head_t *q );
```

---

sleep_on（）函数的作用就是将目前进程的状态置成 TASK_UNINTERRUPTIBLE，并定义一个等待队列，之后把它附属到等待队列头 q，直到资源可获得， q引导的等待队列被唤醒。interruptible_sleep_on（）与 sleep_on（）函数类似，其作用是将目前进程的状态置成 TASK_INTERRUPTIBLE，并定义一个等待队列 q，之后把它附属到等待队列头，直到资源可获得， q引导的等待队列被唤醒或者进程收到信号。

sleep_on（）函数应该与 wake_up（）成对使用，interruptible_sleep_on（）应该与 wake_up_interruptible（）成对使用。

代码清单 10-4和代码清单 10-5分别列出了 sleep_on（）和 interruptible_sleep_on（）函数的源代码。

代码清单 10-4　sleep_on（）函数

```
1 void fastcall _ _sched
interruptible_sleep_on(wait_queue_head_t *q)
2 {
3     SLEEP_ON_VAR
4     /* #define SLEEP_ON_VAR \
5     unsigned long flags; \
6     wait_queue_t wait; \
7     init_waitqueue_entry(&wait, current); */
8     current->state = TASK_UNINTERRUPTIBLE;//改变当前进程状态
9
10     SLEEP_ON_HEAD
11     /* #define SLEEP_ON_HEAD \
12     spin_lock_irqsave(&q->lock,flags); \
13     _ _add_wait_queue(q, &wait); \
14     spin_unlock(&q->lock);*/
15
16     schedule();//放弃CPU
17     SLEEP_ON_TAIL
18     /* #define SLEEP_ON_TAIL \
19     spin_lock_irq(&q->lock); \
20     _ _remove_wait_queue(q, &wait); \
```

```
21      spin_unlock_irqrestore(&q->lock, flags); */
22 }
```

代码清单 10-5　interruptible_sleep_on（）函数

```
1 void fastcall _ _sched
interruptible_sleep_on(wait_queue_head_t *q)
2 {
3      SLEEP_ON_VAR
4
5      current->state = TASK_INTERRUPTIBLE;//
  改变当前进程状态

6
7      SLEEP_ON_HEAD
8      schedule();//放弃CPU
9      SLEEP_ON_TAIL
10 }
```

从代码清单 10-4和代码清单 10-5可以看出，不论是 sleep_on（）还是 interruptible_sleep_on（），其流程都如下所示。

1）定义并初始化一个等待队列，将进程状态改变为 TASK_UNINTERRUPTIBLE（不能被信号打断）或 TASK_INTERRUPTIBLE（可以被信号打断），并将等待队列添加到等待队列头。

2）通过 schedule（）放弃 CPU，调度其他进程执行。

3）进程被唤醒，将等待队列移出等待队列头。

在内核中使用 set_current_state（）函数或 __add_wait_queue（）函数来实现目前进程状态的改变，直接采用 current->state=TASK_UNINTERRUPTIBLE类似的赋值语句也是可行的。通常而言， set_current_state（）函数在任何环境下都可以使用，不会存在并发问题，但是效率要低于 __add_wait_queue（）。

因此，在许多设备驱动中，并不调用 sleep_on（）或 interruptible_sleep_on（），而是直接进行进程的状态改变和切换，如代码清单 10-6所示。

代码清单 10-6　在驱动程序中改变进程状态并调用 schedule（）

```
static ssize_t xxx_write(struct file *file, const char
*buffer,size_t count, loff_t *ppos)
{
    ...
    DECLARE_WAITQUEUE(wait, current); //定义等待队列
    _ _add_wait_queue(&xxx_wait, &wait); //添加等待队列

    ret = count;
    /* 等待设备缓冲区可写*/
    do
    {
        avail = device_writable(...);
        if (avail < 0)
            _ _set_current_state(TASK_INTERRUPTIBLE);//改变进程状
态

        if (avail < 0)
        {
            if (file->f_flags &O_NONBLOCK) //非阻塞
            {
                if (!ret)
                    ret = - EAGAIN;
                goto out;
        }
        schedule(); //调度其他进程执行
        if (signal_pending(current))//如果是因为信号唤醒
        {
            if (!ret)
                ret = - ERESTARTSYS;
            goto out;
        }
    } while (avail < 0);


/* 写设备缓冲区*/
device_write(...)
out:
```

```
remove_wait_queue(&xxx_wait, &wait);//将等待队列移出等待队列头
set_current_state(TASK_RUNNING);//设置进程状态为TASK_RUNNING
return ret;
}
```

10.1.2 支持阻塞操作的 virtualfifo设备驱动

现在我们给 virtualchar加一个设置：把 virtualchar中的全局内存设计成一个 FIFO，只有当 FIFO中有数据的时候（即有进程把数据写到这个 FIFO，而且没有被读进程读空），读进程才能把数据读出，而且读取后的数据会从 virtualchar的全局内存中被拿掉；只有当 FIFO非满时（即还有一些空间未被写，或写满后被读进程从这个 FIFO中读出了数据），写进程才能往这个 FIFO中写入数据。

现在，将 virtualchar重命名为 "virtualfifo"，在 virtualfifo中，读 FIFO将唤醒写 FIFO，而写 FIFO也将唤醒读 FIFO。首先，需要修改设备结构体，在其中增加两个等待队列头，分别对应于读和写，如代码清单 10-7所示。

代码清单 10-7　virtualfifo设备结构体

```
1 struct virtualfifo_dev
2 {
3     struct cdev cdev; /*cdev结构体*/
4     unsigned int current_len; /*fifo有效数据长度*/
5     unsigned char mem[FIFO_SIZE]; /*全局内存*/
6     struct semaphore sem; /*并发控制用的信号量*/
7     wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
8     wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
9 };
```

与 virtualchar设备结构体的另一个不同是，其增加了 current_len成员用于表示目前 FIFO中有效数据的长度。

这个等待队列须在设备驱动模块加载函数中调用 init_waitqueue_head（）被初始化，新的设备驱动模块加载函数如代码清单 10-8所示。

代码清单 10-8　virtualfifo设备驱动模块加载函数

```
1 int virtualfifo_init(void)
2 {
3     int ret;
4     dev_t devno = MKDEV(virtualfifo_major, 0);
5
6     /* 申请设备号*/
7     if (virtualfifo_major)
8         ret = register_chrdev_region(devno, 1,
"virtualfifo");
9     else /* 动态申请设备号*/
10     {
11         ret = alloc_chrdev_region(&devno, 0, 1,
"virtualfifo");
12         virtualfifo_major = MAJOR(devno);
13     }
14     if (ret < 0)
15         return ret;
16     /* 动态申请设备结构体的内存*/
17     virtualfifo_devp = kmalloc(sizeof(struct
virtualfifo_dev),GFP_KERNEL);
18     if (!virtualfifo_devp) /*申请失败*/
19     {
20         ret = - ENOMEM;
21         goto fail_malloc;
22     }
23
24     memset(virtualfifo_devp, 0, sizeof(struct
virtualfifo_dev));
25
26     virtualfifo_setup_cdev(virtualfifo_devp, 0);
27
28     init_MUTEX(&virtualfifo_devp->sem); /*初始化信号量*/
29     init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读
等待队列头*/
30     init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写
等待队列头*/
31
32     return 0;
33
34 fail_malloc: unregister_chrdev_region(devno, 1);
35     return ret;
36 }
```

设备驱动读写操作需要被修改，在读函数中需增加等待virtualfifo_devp->w_wait被唤醒的语句，而在写操作中添加唤醒virtualfifo_devp->r_wait的语句，如代码清单 10-9所示。

代码清单 10-9　增加等待队列后的 virtualfifo读写函数

```
1 /*Virtualfifo读函数*/
2 static ssize_t virtualfifo_read(struct file *filp, char _
_user *buf,size_t
3 count, loff_t *ppos)
4 {
5     int ret;
6     struct virtualfifo_dev *dev = filp->private_data; //获得设备结构体指针
7     DECLARE_WAITQUEUE(wait, current); //定义等待队列
8
9     down(&dev->sem); //获得信号量
10    add_wait_queue(&dev->r_wait, &wait); //进入读等待队列头
11
12    /* 等待FIFO非空*/
13    if (dev->current_len == 0)
14    {
15         if (filp->f_flags &O_NONBLOCK)
16         {
17             ret = - EAGAIN;
18             goto out;
19         }
20         _ _set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
21         up(&dev->sem);
22
23         schedule(); //调度其他进程执行
24         if (signal_pending(current))
25         //如果是因为信号唤醒
26         {
27             ret = - ERESTARTSYS;
28             goto out2;
29         }
30
31         down(&dev->sem);
32     }
33
```

```
34      /* 复制到用户空间*/
35      if (count > dev->current_len)
36          count = dev->current_len;
37
38      if (copy_to_user(buf, dev->mem, count))
39      {
40          ret = - EFAULT;
41          goto out;
42      }
43      else
44      {
45          memcpy(dev->mem, dev->mem + count, dev->current_len
- count);//fifo数据前移
46          dev->current_len -= count; //有效数据长度减少
47          printk(KERN_INFO "read %d
bytes(s),current_len:%d\n", count,dev
48 ->current_len);
49
50      wake_up_interruptible(&dev->w_wait); //唤醒写等待队列
51
52      ret = count;
53 }
54 out: up(&dev->sem); //释放信号量
55 out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队
列头移除
56      set_current_state(TASK_RUNNING);
57      return ret;
58 }
59
60
61 /*Virtualfifo写操作*/
62 static ssize_t virtualfifo_write(struct file *filp, const
char __user *buf,
63 size_t count, loff_t *ppos)
64 {
65      struct virtualfifo_dev *dev = filp->private_data; //获得
设备结构体指针
66 int ret;
67      DECLARE_WAITQUEUE(wait, current); //定义等待队列
68
69      down(&dev->sem); //获取信号量
70      add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头
71
72      /* 等待FIFO非满*/
73      if (dev->current_len == FIFO_SIZE)
74      {
```

```
75          if (filp->f_flags &O_NONBLOCK)
76          //如果是非阻塞访问
77          {
78              ret = - EAGAIN;
79              goto out;
80          }
81          _ _set_current_state(TASK_INTERRUPTIBLE); //改变进程状
态为睡眠
82          up(&dev->sem);
83
84          schedule(); //调度其他进程执行
85          if (signal_pending(current))
86          //如果是因为信号唤醒
87          {
88              ret = - ERESTARTSYS;
89              goto out2;
90          }
91
92          down(&dev->sem); //获得信号量
93      }
94
95      /*从用户空间复制到内核空间*/
96      if (count > GLOBALFIFO_SIZE - dev->current_len)
97          count = GLOBALFIFO_SIZE - dev->current_len;
98
99      if (copy_from_user(dev->mem + dev->current_len, buf,
count))
100      {
101          ret = - EFAULT;
102          goto out;
103      }
104      else
105      {
106          dev->current_len += count;
107          printk(KERN_INFO "written %d
bytes(s),current_len:%d\n",count, dev
108      ->current_len);
109
110      wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
111
112      ret = count;
113      }
114
115      out: up(&dev->sem); //释放信号量
116      out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的
等待队列头移除
```

```
117     set_current_state(TASK_RUNNING);
118     return ret;
119 }
```

代码清单 10-9处理了等待队列进出和进程切换的过程。是否可以把读函数中一大段用于等待 dev->current_len！ =0的内容直接用 wait_event_interruptible（ dev->r_wait， dev->current_len！ =0）替换，把写函数中一大段用于等待 dev->current_len！ =FIFO_SIZE的代码用 wait_event_interruptible（ dev->w_wait， dev->current_len！ =0）替换呢？

实际上，就控制等待队列非空和非满的角度而言，wait_event_interruptible（ dev->r_wait， dev->current_len！ =0）与第 13～ 32行代码的功能完全一样，wait_event_interruptible（ dev->w_wait， dev->current_len！ =0）与第 73～ 93行代码的功能完全一样。细微的区别体现在第 13～ 32行代码和第 73～ 93行代码在进行 schedule（）即切换进程前，通过 up（ &dev->sem）释放了信号量。这一细微的动作意义重大，非如此，则死锁将不可避免。

现在回过来看一下代码清单 10-9的第 15行和第 75行，发现在设备驱动的 read（）、 write（）等功能函数中，可以通过 filp->f_flags标志获得用户空间是否要求非阻塞访问。驱动中可以依据此标志判断用户究竟要求阻塞还是非阻塞访问，从而进行不同的处理。

读者可以通过下面的方式，来验证上面 virtualfifo的设计。

编译 virtualfifo.c并用 insmod命令向 Linux系统加载该模块，成功的话会创建设备文件节点“ /dev/virtualfifo”。之后启动两个进程：一个进程 "cat/dev/virtualfifo&"在后台执行，另一个进程“ echo字符串 /dev/virtualfifo”在前台执行，如下所示：

```
# cat /dev/globalfifo&
# echo 'I want to be' > /dev/virtualfifo
# I want to be
# echo 'DVT for Virtualfifo' > /dev/virtualfifo
# DVT for Virtualfifo
```

每当 echo进程向 /dev/virtualfifo写入一串数据， cat进程就立即将该串数据显现出来。

10.2　Linux的轮询访问

在用户程序中， select（）和 poll（）系统调用也是与设备阻塞与非阻塞访问息息相关的论题。使用非阻塞 I/O的应用程序通常会使用 select（）和 poll（）系统调用查询是否可对设备进行无阻塞的访问。 select（）和 poll（）系统调用最终会引发设备驱动中的 poll（）函数被执行。

select（）和 poll（）系统调用的本质一样，前者在 BSD UNIX中引入，后者在 System V中引入，在 Linux中对它们都提供了支持。

10.2.1　应用程序中的轮询编程

应用程序中最广泛使用的是 BSD UNIX中引入的 select（）系统调用，其原型如下：

```
int select(int numfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
```

其中， readfds、 writefds、 exceptfds分别是被 select（）监视的读、写和异常处理的文件描述符集合， numfds的值是需要检查的号码最高的文件描述符加 1。 timeout参数是一个指向 struct timeval类型的指针，它可以使 select（）在等待 timeout时间后若没有文件描述符准备好则返回。 struct timeval数据结构的定义如代码清单 10-10所示。

代码清单 10-10　timeval结构体定义

```
struct timeval
{
    int tv_sec; /*秒*/
    int tv_usec; /*微秒*/
};
```

下列操作用来设置、清除和判断文件描述符集。

```
FD_ZERO(fd_set *set):清除一个文件描述符集。
FD_SET(int fd,fd_set *set): 将一个文件描述符加入文件描述符集中。
FD_CLR(int fd,fd_set *set): 将一个文件描述符从文件描述符集中清除。
FD_ISSET(int fd,fd_set *set): 判断文件描述符是否被置位。
```

10.2.2 设备驱动中的轮询编程

设备驱动中 poll（）函数的原型如下：

```
unsigned int（*poll）（struct file*filp，struct
poll_table*wait）；
```

第一个参数为 file结构体指针，第二个参数为轮询表指针。这个函数应该进行以下两项工作：

- 对可能引起设备文件状态变化的等待队列调用poll_wait（）函数，将对应的等待队列头添加到poll_table。
- 返回表示是否能对设备进行无阻塞读、写访问的掩码。

用于向 poll_table注册等待队列的 poll_wait（）函数的原型如下：

```
void poll_wait（struct file*filp，wait_queue_heat_t*queue，
poll_table*wait）；
```

---

poll_wait（）函数的名称非常容易让人产生误会，以为它与wait_event（）等一样会阻塞地等待某事件的发生，其实这个函数并不会引起阻塞。 poll_wait（）函数所做的工作是把当前进程添加到wait参数指定的等待列表（ poll_table）中。

驱动程序的 poll（）函数应该返回设备资源的可获取状态，即 POLLIN、 POLLOUT、 POLLPRI、 POLLERR、 POLLNVAL等宏的位“或”结果。每个宏的含义都表明设备的一种状态，如 POLLIN（定义为 0x0001）意味着设备可以无阻塞地读， POLLOUT（定义为 0x0004）意味着设备可以无阻塞地写。

通过以上分析，可得出设备驱动中 poll（）函数的典型模板，如代码清单 10-11所示。

代码清单 10-11　poll（）函数典型模板

```
static unsigned int xxx_poll(struct file *filp, poll_table
*wait)
{
    unsigned int mask = 0;
    struct xxx_dev *dev = filp->private_data; /*获得设备结构体指针
*/

    ...
    poll_wait(filp, &dev->r_wait, wait);//加读等待队列头
    poll_wait(filp, &dev->w_wait, wait);//加写等待队列头

    if (...)//可读
    {
        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
    }

    if (...)//可写
    {
        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
    }

    ...
    return mask;
}
```

10.2.3　支持轮询操作的 virtualfifo驱动

1.在virtualfifo驱动中增加轮询操作

在 virtualfifo的 poll（）函数中，首先将设备结构体中的 r_wait和 w_wait等待队列头添加到等待列表，然后通过判断 dev->current_len是否等于 0来获得设备的可读状态，通过判断 dev->current_len是否等于 FIFO_SIZE来获得设备的可写状态，如代码清单 10-12所示。

代码清单 10-12　virtualfifo设备驱动的 poll（）函数

```
static unsigned int virtualfifo_poll(struct file *filp,
poll_table *wait)
{
    unsigned int mask = 0;
    struct virtualfifo_dev *dev = filp->private_data; /*获得设备
结构体指针*/

    down(&dev->sem);

    poll_wait(filp, &dev->r_wait, wait);
    poll_wait(filp, &dev->w_wait, wait);
    /*fifo非空*/
    if (dev->current_len != 0)
    {
        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
    }
    /*fifo非满*/
    if (dev->current_len != FIFO_SIZE)
    {
        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
    }

    up(&dev->sem);
    return mask;
}
```

注意要把 virtualfifo_poll赋给 virtualfifo_fops的 poll成员，如下所示：

```
static const struct file_operations virtualfifo_fops =
{
```

```
    ...
    .poll = virtualfifo_poll,
    ...
};
```

2.在用户空间验证virtualfifo设备的轮询

编写一个应用程序 pollmonitor.c用于监控 virtualfifo设备的可读写状态，这个程序如代码清单 10-13所示。

代码清单 10-13　监控 virtualfifo是否可非阻塞读写的应用程序

```
1 #include ...
2
3 #define FIFO_CLEAR 0x1
4 #define BUFFER_LEN 20
5 main()
6 {
7     int fd, num;
8     char rd_ch[BUFFER_LEN];
9     fd_set rfds,wfds; //读/写文件描述符集
10
11     /*以非阻塞方式打开/dev/virtualfifo设备文件*/
12     fd = open("/dev/virtualfifo", O_RDONLY | O_NONBLOCK);
13     if (fd != - 1)
14     {
15         /*FIFO清零*/
16         if (ioctl(fd, FIFO_CLEAR, 0) < 0)
17         {
18             printf("ioctl command failed\n");
19         }
20         while (1)
21         {
22             FD_ZERO(&rfds);
23             FD_ZERO(&wfds);
24             FD_SET(fd, &rfds);
25             FD_SET(fd, &wfds);
26
27             select(fd + 1, &rfds, &wfds, NULL, NULL);
28             /*数据可获得*/
29             if (FD_ISSET(fd, &rfds))
30             {
```

```
31                  printf("Poll monitor:can be read\n");
32              }
33              /*数据可写入*/
34              if (FD_ISSET(fd, &wfds))
35              {
36                  printf("Poll monitor:can be written\n");
37              }
38          }
39      }
40      else
41      {
42          printf("Device open failure\n");
43      }
44  }
```

运行时看到，当没有任何输入即 FIFO为空时，程序不断地输出 "Poll monitor:can be written"；当通过 echo向 /dev/virtualfifo写入一些数据后，将输出 "Poll monitor:can be read"和 "Poll monitor:can be written"；如果不断地通过 echo向 /dev/virtualfifo写入数据直至写满 FIFO，发现 pollmonitor程序将只输出 "Poll monitor:can be read"。对于 virtualfifo设备而言，不会出现既不能读又不能写的情况。

第11章　Linux设备驱动中的异步访问

在设备驱动中使用异步通知，使得设备的访问条件满足时，可由驱动主动通知应用程序进行访问。这样，使用非阻塞I/O的应用程序无须轮询设备是否可访问，而阻塞访问也可以被类似“中断”的异步通知所取代。

下面首先讲解异步通知的概念和作用，接着基于Linux 2.6中的AIO展开阐述；最后增加异步通知的virtualfifo驱动为例子，体验异步访问真谛。

11.1　Linux 2.6中的异步访问

11.1.1　异步访问概念与 GNU C库函数

Linux系统中最常用的输入 /输出（ I/O）模型是同步 I/O，前面第 7～ 10章中我们就是基于这个模型来实现设备 I/O的。在这个模型中，当请求发出之后，应用程序就会阻塞，直到请求满足为止。这是很好的一种解决方案，因为调用应用程序在等待 I/O请求完成时不需要使用任何 CPU。但是在某些情况下， I/O请求期间可能需要与其他进程产生交叠，以完成其他工作任务。可移植操作系统接口（ POSIX）异步访问（又常称为异步 I/O， AIO）应用程序接口（ API）就提供了这种功能。

事实上，阻塞与非阻塞访问、 poll（）函数提供了较好的解决设备访问的机制，但是如果有了异步通知整套机制就更加完整了：应用程序在申请访问设备的同时，可以去完成别的任务；当设备状态变化时，可发出异步通知，使得应用程序即时地访问设备。 Linux异步 I/O是2.6版本内核的一个标准特性，但是我们在 2.4版本内核的补丁中也可以找到它。 AIO基本思想是允许进程发起很多 I/O操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O操作完成的通知时，进程就可以检索 I/O操作的结果。

异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点类似于硬件上“中断”的概念，比较准确的称谓是“信号驱动的异步 I/O”。信号是在软件层次上对中断机制的一种模拟。在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，这样一个进程不必通过任何操作来等待信号的到达，而进程也不知道信号到底什么时候到达。

阻塞 I/O意味着一直等待设备可访问后再访问，非阻塞 I/O中使用 poll（）意味着查询设备是否可访问，而异步通知则意味着设备通知自身可访问，实现了异步 I/O。由此可见，这几种 I/O方式可以互为补充。

图 11-1呈现了阻塞 I/O、结合 poll（）的非阻塞 I/O及异步 I/O在时间先后顺序上的不同。

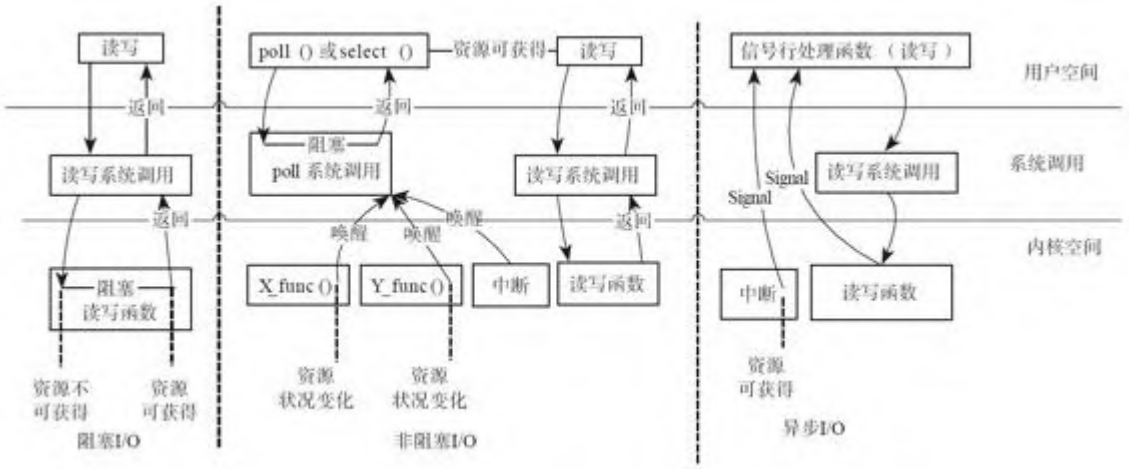


图11-1　阻塞I/O、非阻塞I/O、异步I/O的区别

阻塞 I/O、非阻塞 I/O和异步通知之间本身没有优劣，我们应该根据不同的应用场景，灵活合理地选择。

注： select（）函数所提供的功能非阻塞 I/O与 AIO都有进程与底层硬件操作异步的特点，但它是针对通知事件进行阻塞，而不是对 I/O调用进行阻塞；通过图 11-1我们应该更容易理解它们之间的区别。

在异步 I/O中，我们可以同时发起多个传输操作。这需要每个传输操作都有唯一的上下文，这样才能在它们完成时区分到底是哪个传输操作完成了。在 AIO中，通过 aiocb（ AIO I/O control block）结构体进行区分。这个结构体包含了有关传输的所有信息，包括为数据准备的用户缓冲区。在产生 I/O（称为完成）通知时， aiocb结构体就被用来唯一标识所完成的 I/O操作。

AIO系列 API被 GNU C库函数所包含，它被 POSIX.1b所要求，主要包括如下函数。

1.aio_read

aio_read（）函数请求对一个有效的文件描述符进行异步读操作。这个文件描述符可以表示一个文件、套接字甚至管道。 aio_read函数的原型如下：

```
int aio_read（struct aiocb*aiocbp）;
```

aio_read（）函数在请求进行排队之后会立即返回。如果执行成功，返回值就为 0；如果出现错误，返回值就为 -1，并设置系统 errno的值。

2.aio_write

aio_write（）函数用来请求一个异步写操作，其函数原型如下：

```
int aio_write（struct aiocb*aiocbp）;
```

aio_write（）函数会立即返回，说明请求已经进行排队；成功时返回值为 0，失败时返回值为 -1，并相应地设置系统 errno。

3.aio_error

aio_error函数被用来确定请求的状态，其函数原型如下：

```
int aio_error（struct aiocb*aiocbp）;
```

这个函数可以返回以下内容：

EINPROGRESS：说明请求尚未完成。

ECANCELLED：说明请求被应用程序取消了。

-1：说明发生了错误，具体错误原因由系统 errno记录。

4.aio_return

异步 I/O和标准块 I/O之间的另外一个区别是不能立即访问读 /写函数的返回状态，因为并没有阻塞在 read（）等调用上。在标准的 read（）等调用中，返回状态是在这些函数返回时提供的。但是在异步 I/O中，我们要使用 aio_return（）函数。这个函数的原型如下：

```
ssize_t aio_return（struct aiocb*aiocbp）;
```

只有在 aio_error（）调用确定请求已经完成（可能成功，也可能发生了错误）之后，才会调用这个函数。 aio_return（）的返回值就等价于同步情况中 read（）或 write（）系统调用的返回值（所传输的字节数，如果发生错误返回值就为 -1）。

代码清单 11-1给出了用户空间应用程序进行异步读操作的一个例程，它首先打开文件，然后准备 aiocb结构体，之后调用 aio_read（ &my_aiocb）提出异步读请求，当 aio_error（ &my_aiocb）==EINPROGRESS即操作还在进行中时，一直等待，结束后通过 aio_return（ &my_aiocb）获得返回值。

代码清单 11-1　用户空间异步读例程

```
#include <aio.h>
...
int fd, ret;
struct aiocb my_aiocb;

fd = open("file.txt", O_RDONLY);
if (fd < 0)
    perror("open");

/*清零aiocb结构体*/
bzero((char*) &my_aiocb, sizeof(struct aiocb));
/*为aiocb请求分配数据缓冲区*/
my_aiocb.aio_buf = malloc(BUFSIZE + 1);
if (!my_aiocb.aio_buf)
    perror("malloc");

/*初始化aiocb的成员*/
```

```
my_aiocb.aio_fildes = fd;
my_aiocb.aio_nbytes = BUFSIZE;
my_aiocb.aio_offset = 0;

ret = aio_read(&my_aiocb);
if (ret < 0)
    perror("aio_read");

while (aio_error(&my_aiocb) == EINPROGRESS)
;

if ((ret = aio_return(&my_iocb)) > 0)
{
    /*获得异步读的返回值*/
}
else
{
    /*读失败,分析errorno */
}
```

用户可以使用 aio_suspend（）函数来挂起（或阻塞）调用进程，直到异步请求完成为止，此时会产生一个信号，或者发生其他超时操作。调用者提供了一个 aiocb引用链表，其中任何一个完成都会导致aio_suspend（）返回。 aio_suspend的函数原型如下：

```
int aio_suspend（const struct aiocb*const cblist[]，int n，const struct timespec*timeout）；
```

代码清单 11-2给出了用户空间异步读操作时使用 aio_suspend（）函数的例子。

代码清单 11-2　用户空间异步 I/0的 aio_suspend（）函数使用例程

```
struct aioct *cblist[MAX_LIST]
/*清零aioct结构体链表*/
bzero((char *)cblist, sizeof(cblist));
/*将一个或更多的aiocb放入aioct结构体链表*/
cblist[0] = &my_aiocb;
```

```
ret = aio_read( &my_aiocb );
ret = aio_suspend( cblist, MAX_LIST, NULL );
```

---

aio_cancel（）函数允许用户取消对某个文件描述符执行的一个或所有 I/O请求。其原型如下：

```
int aio_cancel（int fd，struct aiocb*aiocbp）；
```

如果要取消一个请求，用户需提供文件描述符和 aiocb引用。如果这个请求被成功取消了，那么这个函数就会返回 AIO_CANCELED。如果请求完成了，这个函数就会返回 AIO_NOTCANCELED。

如果要取消对某个给定文件描述符的所有请求，用户需要提供这个文件的描述符以及一个对 aiocbp指针参数的 NULL引用。如果所有的请求都取消了，这个函数就会返回 AIO_CANCELED；如果至少有一个请求没有被取消，那么这个函数就会返回 AIO_NOT_CANCELED；如果没有一个请求可以被取消，那么这个函数就会返回 AIO_ALLDONE。然后，可以使用 aio_error（）来验证每个 AIO请求，如果某请求已经被取消了，那么 aio_error（）就会返回 -1，并且 errno会被设置为 ECANCELED。

lio_listio（）函数可用于同时发起多个传输。这个函数非常重要，它使得用户可以在一个系统调用（一次内核上下文切换）中启动大量的 I/O操作。 lio_listio API函数的原型如下：

```
int lio_listio（int mode，struct aiocb*list[]，int nent，struct sigevent*sig）；
```

---

mode参数可以是 LIO_WAIT或 LIO_NOWAIT。 LIO_WAIT会阻塞这个调用，直到所有的 I/O都完成为止。在操作进行排队之后， LIO_NOWAIT就会返回。 list是一个 aiocb引用的列表，最大元素的个数是由 nent定义的。如果 list的元素为 NULL， lio_listio（）会将其忽略。

代码清单 11-3给出了用户空间异步 I/O操作时使用 lio_listio（）函数的例子。

代码清单 11-3　用户空间异步 I/O的 lio_listio（）函数使用例程

```
1 struct aiocb aiocb1, aiocb2;
2 struct aiocb *list[MAX_LIST];
3 ...
4 /*准备第一个aiocb */
5 aiocb1.aio_fildes = fd;
6 aiocb1.aio_buf = malloc( BUFSIZE+1 );
7 aiocb1.aio_nbytes = BUFSIZE;
8 aiocb1.aio_offset = next_offset;
9 aiocb1.aio_lio_opcode = LIO_READ; /*异步读操作*/
10 ... /*准备多个aiocb */
11 bzero( (char *)list, sizeof(list) );
12
13 /*将aiocb填入链表*/
14 list[0] = &aiocb1;
15 list[1] = &aiocb2;
16 ...
17 ret = lio_listio( LIO_WAIT, list, MAX_LIST, NULL );/*发起大量I/O操作*/
```

上述代码第 9行中，因为是进行异步读操作，所以操作码为LIO_READ，对于写操作来说，应该使用 LIO_WRITE作为操作码，而LIO_NOP意味着空操作。

11.1.2　使用信号作为异步访问的通知

第 3章中讲述的信号作为异步通知的机制在 AIO中仍然是适用的，为使用信号，使用 AIO的应用程序同样需要定义信号处理程序，在指定的信号被产生时会触发调用这个处理程序。作为信号上下文的一部分，特定的 aiocb请求被提供给信号处理函数用来区分 AIO请求。

代码清单 11-4给出了使用信号作为 AIO异步 I/O通知机制的例子。

代码清单 11-4　使用信号作为 AIO异步 I/O通知机制例程

```
1 /*设置异步I/O请求*/
2 void setup_io(...)
3 {
4      int fd;
5      struct sigaction sig_act;
6      struct aiocb my_aiocb;
7      ...
8      /*设置信号处理函数*/
9      sigemptyset(&sig_act.sa_mask);
10     sig_act.sa_flags = SA_SIGINFO;
11     sig_act.sa_sigaction = aio_completion_handler;
12
13  /*设置AIO请求*/
14     bzero((char*) &my_aiocb, sizeof(struct aiocb));
15     my_aiocb.aio_fildes = fd;
16     my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
17     my_aiocb.aio_nbytes = BUF_SIZE;
18     my_aiocb.aio_offset = next_offset;
19
20     /*连接AIO请求和信号处理函数*/
21     my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
22     my_aiocb.aio_sigevent.sigev_signo = SIGIO;
23     my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
24
25     /*将信号与信号处理函数绑定*/
26     ret = sigaction(SIGIO, &sig_act, NULL);
27     ...
28     ret = aio_read(&my_aiocb); /*发出异步读请求*/
29 }
30
31 /*信号处理函数*/
32 void aio_completion_handler(int signo, siginfo_t *info, void
*context)
33 {
34     struct aiocb *req;
35
36     /*确定是我们需要的信号*/
37     if (info->si_signo == SIGIO)
38     {
39         req = (struct aiocb*)info->si_value.sival_ptr; /*获得
aiocb*/
40
41         /*请求的操作完成了吗? */
42         if (aio_error(req) == 0)
43         {
```

```
44                 /*请求的操作完成,获取返回值*/
45                 ret = aio_return(req);
46             }
47         }
48         return ;
49 }
```

特别要注意上述代码的第 39行通过（ struct aiocb*） info->si_value.sival_ptr获得了信号对应的 aiocb。

11.1.3 使用回调函数作为异步访问的通知

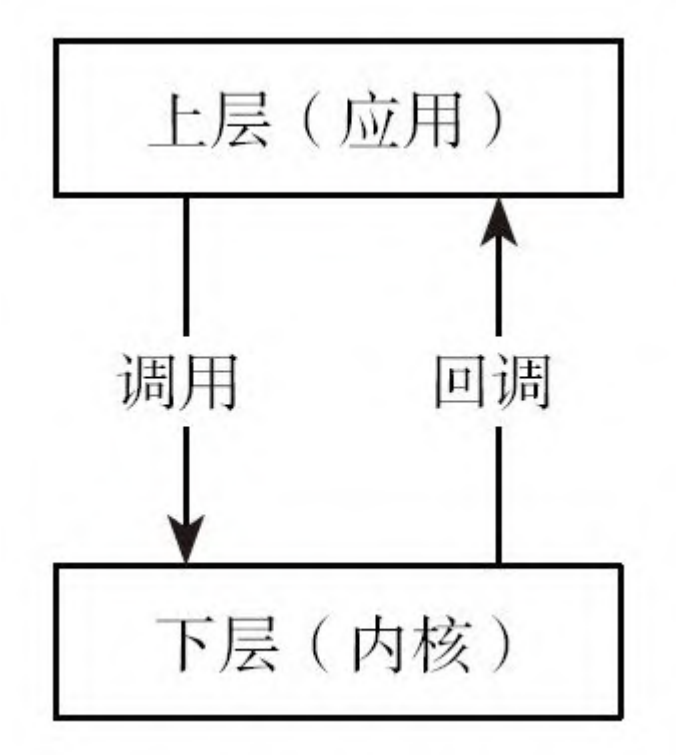


图11-2 回调与调用

除了信号之外，应用程序还可提供一个回调（ Callback）函数给内核，以便 AIO的请求完成后内核调用这个函数。

一般来说，我们将下层对上层（如内核对应用）的调用都称为“回调”，而上层对下层（如进行 Linux系统调用）的调用称为“调用”，如图 11-2所示。

代码清单 11-5给出了使用回调函数作为 AIO异步 I/O通知机制的例子。

代码清单 11-5　使用回调函数作为 AIO异步 I/O通知机制例程

```
/*设置异步I/O请求*/
void setup_io(...)
{
    int fd;
    struct aiocb my_aiocb;
    ...
    /*设置AIO请求*/
    bzero((char*) &my_aiocb, sizeof(struct aiocb));
    my_aiocb.aio_fildes = fd;
    my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
    my_aiocb.aio_nbytes = BUF_SIZE;
    my_aiocb.aio_offset = next_offset;

    /*连接AIO请求和线程回调函数*/
    my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
    my_aiocb.aio_sigevent.notify_function =
aio_completion_handler;
    /*设置回调函数*/
    my_aiocb.aio_sigevent.notify_attributes = NULL;
    my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
    ... ret = aio_read(&my_aiocb); //发起AIO请求
}

/*异步I/O完成回调函数*/
void aio_completion_handler(sigval_t sigval)
{
    struct aiocb *req;
    req = (struct aiocb*)sigval.sival_ptr;
    /* AIO请求完成? */
    if (aio_error(req) == 0)
{
        /*请求完成,获得返回值*/
        ret = aio_return(req);
    }

    return ;
}
```

上述程序在创建 aiocb请求之后，使用 SIGEV_THREAD请求了一个线程回调函数来作为通知方法。在回调函数中，通过（ struct aiocb*）sigval.sival_ptr可以获得对应的 aiocb指针，使用 AIO函数可验证请求是否已经完成。

proc文件系统包含了两个虚拟文件，它们可以用来对异步 I/O的性能进行优化。

·/proc/sys/fs/aio-nr文件提供了系统范围异步I/O请求的数目。

·/proc/sys/fs/aio-max-nr文件是所允许的并发请求的最大个数，这个值通常是65536，这对于大部分应用程序来说已经足够了。

11.1.4 异步访问与设备驱动

在内核中，每个 I/O请求都对应于一个 kiocb结构体，其 ki_filp成员指向对应的 file指针，通过 is_sync_kiocb（）可以判断某 kiocb是否为同步 I/O请求，如果返回非真，表示为异步 I/O请求。

块设备和网络设备本身是异步的，只有字符设备必须明确表明应支持AIO。 AIO对于大多数字符设备而言都不是必需的，只有极少数设备需要。比如，对于磁带机，由于 I/O操作很慢，这时候使用异步 I/O将改善性能。

字符设备驱动程序中， file_operations包含 3个与 AIO相关的成员函数，如下所示：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer, size_t
count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

aio_read（）和 aio_write（）与 file_operations中的 read（）和 write（）中的 offset参数不同， aio_xxx（）是直接传递值，而后者传递的是指针，这是因为 AIO从来不需要改变文件的位置。

aio_read（）和 aio_write（）函数本身不一定完成了读和写操作，它只是发起、初始化读和写操作，代码清单 11-6给出了驱动程序中aio_read（）和 aio_write（）函数的实现例子。

代码清单 11-6　设备驱动中的异步 I/O函数

```
1 /*异步读*/
2 static ssize_t xxx_aio_read(struct kiocb *iocb, char *buf,
size_t count, loff_t pos)
3 {
4     return xxx_defer_op(0, iocb, buf, count, pos);
5 }
6
7 /*异步写*/
8 static ssize_t xxx_aio_write(struct kiocb *iocb, const char
*buf,size_t count, loff_t pos)
9 {
10     return xxx_defer_op(1, iocb, (char*)buf, count, pos);
11 }
12
13 /*初始化异步I/O*/
14 static int xxx_defer_op(int write, struct kiocb *iocb, char
*buf,size_t count, loff_t pos)
15 {
16     struct async_work *async_wk;
17     int result;
18     /*当可以访问buffer时进行复制*/
19     if (write)
20         result = xxx_write(iocb->ki_filp, buf, count, &pos);
21     else
22         result = xxx_read(iocb->ki_filp, buf, count, &pos);
23     /*如果是同步IOCB,立即返回状态*/
24     if (is_sync_kiocb(iocb))
25         return result;
26
27     /*否则,推后几μs执行*/
28     async_wk = kmalloc(sizeof(*async_wk), GFP_KERNEL);
29     if (async_wk == NULL)
30         return result;
31     /*调度延迟的工作*/
32     async_wk->iocb = iocb;
33     async_wk->result = result;
34     INIT_WORK(&async_wk->work, xxx_do_deferred_op,
```

```
async_wk);
35      schedule_delayed_work(&async_wk->work, HZ / 100);
36      return - EIOCBQUEUED; /*控制权返回用户空间*/
37 }
38
39 /*延迟后执行*/
40 static void xxx_do_deferred_op(void *p)
41 {
42      struct async_work *async_wk = (struct async_work*)p;
43      aio_complete(async_wk->iocb, async_wk->result, 0);
44      kfree(async_wk);
45 }
```

上述代码中最核心的是使用 async_work（异步工作）结构体将操作延后执行， async_work结构体定义如代码清单 11-7所示，通过 schedule_delayed_work（）函数可以调度其执行。第 43行对 aio_complete（）的调用用于通知内核驱动程序已经完成了操作。

代码清单 11-7　async_work结构体

```
struct async_work
{
    struct kiocb *iocb; //kiocb结构体指针
    int result; //执行结果
    struct work_struct work; //工作结构体
};
```

11.2 异步 Fifo驱动例子

这是我们关于 Linux字符设备驱动的最后一个知识点，也通过例子来介绍，相信读者会更容易理解。

11.2.1 在 virtualfifo驱动中增加异步通知

首先，将异步结构体指针添加到 virtualfifo_dev设备结构体内，如代码清单 11-8所示。

代码清单 11-8 增加异步通知后的 virtualfifo设备结构体

```
struct virtualfifo_dev
{
    struct cdev cdev; /*cdev结构体*/
    unsigned int current_len; /*fifo有效数据长度*/
    unsigned char mem[FIFO_SIZE]; /*全局内存*/
    struct semaphore sem; /*并发控制用的信号量*/
    wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
    wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
    struct fasync_struct *async_queue; /* 异步结构体指针,用于读*/
};
```

同时在 virtualfifo驱动中，实现 fasync（）功能函数，以响应上层对设备的异步访问。这个函数如代码清单 11-9所示。

代码清单 11-9 支持异步通知的 virtualfifo设备驱动的 fasync（）函数

```
static int virtualfifo_fasync(int fd, struct file *filp, int
mode)
{
    struct virtualfifo_dev *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

在 virtualfifo设备被正确写入之后，它变得可读，这个时候驱动应释放 SIGIO信号以便应用程序捕获。代码清单 11-10给出了支持异步通知的 virtualfifo设备驱动的写函数。

代码清单 11-10　支持异步通知的 virtualfifo设备驱动的写函数

```
static ssize_t virtualfifo_write(struct file *filp, const char
__user *buf, size_t count, loff_t *ppos)
{
    struct virtualfifo_dev *dev = filp->private_data; //获得设备
结构体指针
    int ret;
    DECLARE_WAITQUEUE(wait, current); //定义等待队列

    down(&dev->sem); //获取信号量
    add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头

    /* 等待FIFO非满*/
    if (dev->current_len == GLOBALFIFO_SIZE)
    {
        if (filp->f_flags &O_NONBLOCK)
        //如果是非阻塞访问
        {
            ret = - EAGAIN;
            goto out;
        }
        _ _set_current_state(TASK_INTERRUPTIBLE); //改变进程状态
为睡眠
        up(&dev->sem);

        schedule(); //调度其他进程执行
        if (signal_pending(current))
        //如果是因为信号唤醒
        {
            ret = - ERESTARTSYS;
            goto out2;
        }

        down(&dev->sem); //获得信号量
    }
```

```
    /*从用户空间复制到内核空间*/
    if (count > FIFO_SIZE - dev->current_len)
        count = FIFO_SIZE - dev->current_len;

    if (copy_from_user(dev->mem + dev->current_len, buf,
count))
    {
        ret = - EFAULT;
        goto out;
    }
    else
    {
        dev->current_len += count;
        printk(KERN_INFO "written %d
bytes(s),current_len:%d\n", count, dev->current_len);

        wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
        /* 产生异步读信号*/
        if (dev->async_queue)
            kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
        ret = count;
    }

out: up(&dev->sem); //释放信号量
out2:remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列头
移除
    set_current_state(TASK_RUNNING);
    return ret;
}
```

另外，增加异步通知后的 virtualfifo设备驱动的 release（）函数中，需调用 virtualfifo_fasync（）函数将文件从异步通知列表中删除，代码清单 11-11给出了支持异步通知的 virtualfifo_release（）函数。

代码清单 11-11　增加异步通知后的 virtualfifo设备驱动的 release（）函数

```
int virtualfifo_release(struct inode *inode, struct file *filp)
{
    struct virtualfifo_dev *dev = filp->private_data;
    /* 将文件从异步通知列表中删除*/
```

```
    globalfifo_fasync( - 1, filp, 0);
    return 0;
}
```

11.2.2 在用户空间验证 virtualfifo的异步通知

现在，我们可以编写一个在用户空间验证 virtualfifo异步通知的程序，这个程序在接收到由 virtualfifo发出的信号后将输出信号值，如代码清单 11-12所示。

代码清单 11-12 监控 virtualfifo异步通知信号的应用程序

```
#include ...

/*接收到异步读信号后的动作*/
void input_handler(int signum)
{
    printf("receive a signal from
virtualfifo,signalnum:%d\n",signum);
}

main()
{
    int fd, oflags;
    fd = open("/dev/virtualfifo", O_RDWR, S_IRUSR | S_IWUSR);
    if (fd != - 1)
    {
        //启动信号驱动机制
        signal(SIGIO, input_handler); //让input_handler()处理
SIGIO信号
        fcntl(fd, F_SETOWN, getpid());
        oflags = fcntl(fd, F_GETFL);
        fcntl(fd, F_SETFL, oflags | FASYNC);
        while(1)
        {
            sleep(100);
        }
    }
    else
    {
        printf("device open failure\n");
```

```
    }
}
```

加载新的 virtualfifo设备驱动并创建设备文件节点，运行上述程序后，每当通过 echo向 /dev/virtualfifo写入新的数据时，input_handler（）将会被调用，如下所示：

```
# echo 0 > /dev/virtualfifo
# receive a signal from globalfifo,signalnum:29
```

第12章　Linux块设备驱动

通过前面的学习，大家已对Linux驱动模型与char类型驱动有所了解。

而块设备是与字符设备并列的概念，这两类设备在Linux中的驱动结构有较大差异，总体而言，块设备驱动比字符设备驱动要复杂得多，在I/O操作上表现出极大的不同。缓冲、I/O调度、请求队列等都是与块设备驱动相关的概念。本章将讲解Linux块设备驱动的编程方法。

当然有了前面的基础，读者可以快速地通读该章。不妨在遇到具体开发时，再回头仔细阅读。

12.1　块设备的 I/O操作特点

字符设备与块设备 I/O操作的不同如下：

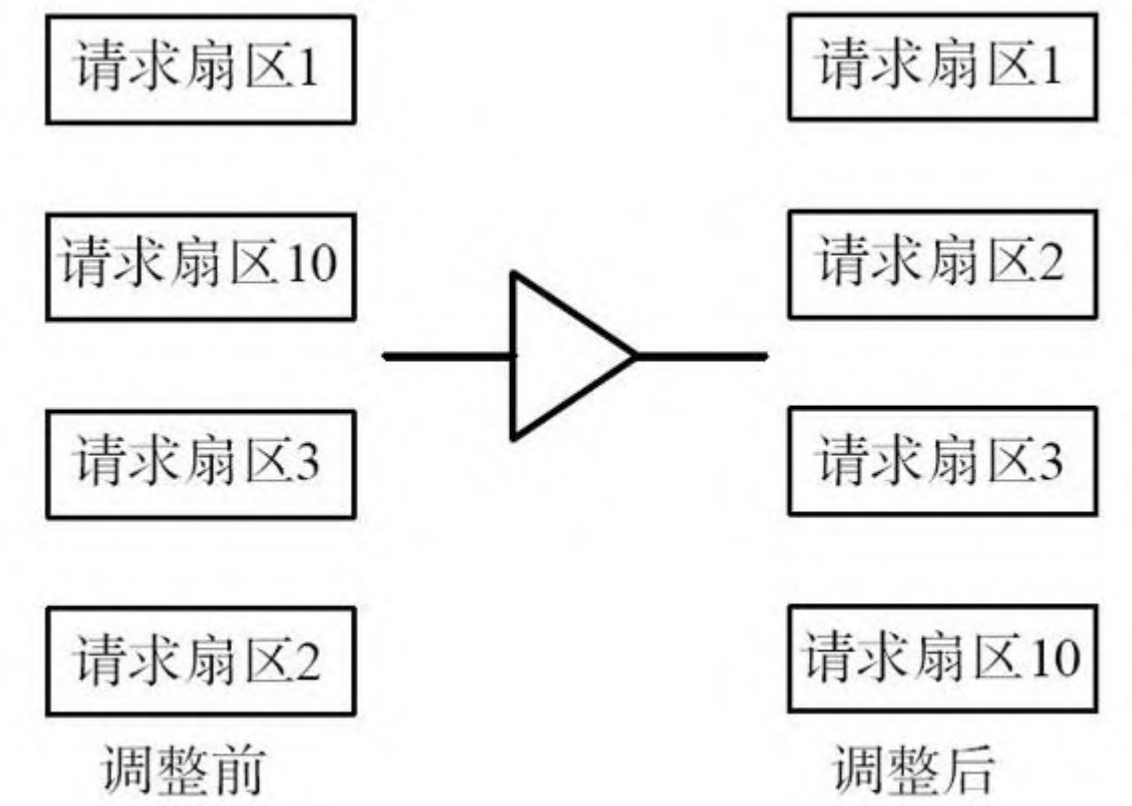


图12-1　调整块设备I/O操作的顺序

1）块设备只能以块为单位接受输入和返回输出，而字符设备则以字节为单位。大多数设备是字符设备，因为它们不需要缓冲而且不以固定块大小进行操作。

2）块设备对于 I/O请求有对应的缓冲区，因此它们可以选择以什么顺序进行响应，字符设备无须缓冲且被直接读写。对于存储设备而言调整读写的顺序作用巨大，因为读写连续的扇区比分离的扇区更快。

3）字符设备只能被顺序读写，而块设备可以随机访问。虽然块设备可随机访问，但是对于磁盘这类机械设备而言，顺序地组织块设备的访问可以提高性能，如图 12-1所示，对扇区 1、 10、 3、 2的请求被

调整为对扇区 1、 2、 3、 10的请求。而对 SD卡、 RAMDisk等块设备而言，因为不存在机械上的原因，进行这样的调整就没有必要。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

12.2 Linux块设备驱动结构

12.2.1 block_device_operations结构体

在块设备驱动中，有一个类似于字符设备驱动中 file_operations结构体的 block_device_operations结构体，它是对块设备操作的集合，定义如代码清单 12-1所示。

代码清单 12-1 block_device_operations结构体

```
struct block_device_operations
{
    int(*open)(struct inode *, struct file*); //打开
    int(*release)(struct inode *, struct file*); //释放
    int(*ioctl)(struct inode *, struct file *, unsigned,
unsigned long);//ioctl
    long(*unlocked_ioctl)(struct file *, unsigned, unsigned
long);
    long(*compat_ioctl)(struct file *, unsigned, unsigned
long);
    int(*direct_access)(struct block_device *, sector_t,
unsigned long*);
    int(*media_changed)(struct gendisk*); //介质被改变?
    int(*revalidate_disk)(struct gendisk*); //使介质有效
    int(*getgeo)(struct block_device *, struct hd_geometry*);//
填充驱动器信息
    struct module *owner; //模块拥有者
};
```

下面对其主要的成员函数进行分析。

1.打开和释放

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

与字符设备驱动类似，当设备被打开和关闭时将分别调用它们。

2.IO控制

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned
int cmd, unsigned long arg);
```

上述函数是 ioctl（）系统调用的实现，块设备包含大量的标准请求，这些标准请求由 Linux块设备层处理，因此大部分块设备驱动的ioctl（）函数相当短。

3.介质改变

```
int（*media_changed）（struct gendisk*gd）；
```

被内核调用来检查是否驱动器中的介质已经改变，如果是，则返回一个非 0值，否则返回 0。这个函数仅适用于支持可移动介质的驱动器，通常需要在驱动中增加一个表示介质状态是否改变的标志变量，非可移动设备的驱动不需要实现这个方法。

4.使介质有效

```
int（*revalidate_disk）（struct gendisk*gd）；
```

---

revalidate_disk（）函数被调用来响应一个介质改变，它给驱动一个机会，为新介质准备好进行必要的初始工作。

5.获得驱动器信息

```
int（*getgeo）（struct block_device*，struct hd_geometry*）；
```

该函数根据驱动器的几何信息填充一个 hd_geometry结构体，hd_geometry结构体包含磁头、扇区、柱面等信息。

6.模块指针

```
struct module*owner;
```

一个指向拥有这个结构体的模块的指针，它通常被初始化为THIS_MODULE。

12.2.2　gendisk结构体

在 Linux内核中，使用 gendisk（通用磁盘）结构体来表示一个独立的磁盘设备（或分区），这个结构体的定义如代码清单 12-2所示。

代码清单 12-2　gendisk结构体

```
struct gendisk
{
    int major; /* 主设备号 */
    int first_minor; /*第1个次设备号*/
    int minors; /* 最大的次设备数,如果不能分区,则为1*/
    char disk_name[32]; /* 设备名称 */
    struct hd_struct **part; /* 磁盘上的分区信息 */
    struct block_device_operations *fops; /*块设备操作结构体*/
    struct request_queue *queue; /*请求队列*/
    void *private_data; /*私有数据*/
    sector_t capacity; /*扇区数,512字节为1个扇区*/

    int flags;
    char devfs_name[64];
    int number;
    struct device *driverfs_dev;
    struct kobject kobj;

    struct timer_rand_state *random;
```

```
    int policy;

    atomic_t sync_io; /* RAID */
    unsigned long stamp;
    int in_flight;
#ifdef CONFIG_SMP
    struct disk_stats *dkstats;
#else
    struct disk_stats dkstats;
#endif
};
```

---

major、 first_minor和 minors共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而次设备号则不同。 fops为block_device_operations结构体，即上一节描述的块设备操作集合。queue是内核用来管理这个设备的 I/O请求队列的指针。 capacity表明设备的容量，以 512个字节为单位。 private_data可用于指向磁盘的任何私有数据，用法与字符设备驱动 file结构体的 private_data类似。

Linux内核提供了一组函数来操作 gendisk，如下所示。

1.分配 gendisk

gendisk结构体是一个动态分配的结构体，它需要特别的内核操作来初始化，驱动不能自己分配这个结构体，而应该使用下列函数来分配gendisk：

```
struct gendisk*alloc_disk（int minors）;
```

---

minors参数是这个磁盘使用的次设备号的数量，一般也就是磁盘分区的数量，此后 minors不能被修改。

2.增加 gendisk

gendisk结构体被分配之后，系统还不能使用这个磁盘，需要调用如下函数来注册这个磁盘设备。

```
void add_disk（struct gendisk*gd）；
```

特别要注意的是，对 add_disk（）的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3.释放 gendisk

当不再需要一个磁盘时，应当使用如下函数释放 gendisk。

```
void del_gendisk（struct gendisk*gd）；
```

4.gendisk引用计数

gendisk中包含一个 kobject成员，因此，它是一个可被引用计数的结构体。通过 get_disk（）和 put_disk（）函数可用来操作引用计数，这个工作一般不需要驱动亲自做。通常对 del_gendisk（）的调用会去掉 gendisk的最终引用计数，但是这一点并不是必须的。因此，在 del_gendisk（）被调用后，这个结构体可能继续存在。

5.设置 gendisk容量

```
void set_capacity（struct gendisk*disk，sector_t size）；
```

块设备中最小的可寻址单元是扇区，扇区大小一般是 2的整数倍，最常见的大小是 512字节。扇区的大小是设备的物理属性，扇区是所有块设备的基本单元，块设备无法对比它还小的单元进行寻址和操作，但可以扇区的整数倍来实施块设备访问。虽然大多数块设备的扇区大小都是 512字节，不过其他大小的扇区也很常见，比如，很多 CD-ROM盘的扇区都是 2KB。不管物理设备的真实扇区大小是多少，内核与块设备驱动交互的扇区都以 512字节为单位。因此， set_capacity（）函数也以 512字节为单位。

12.2.3 request与 bio结构体

1.请求

在 Linux块设备驱动中，使用 request结构体来表征等待进行的 I/O请求，这个结构体的定义如代码清单 12-3所示。

代码清单 12-3 request结构体

```
struct request
{
    struct list_head queuelist; /*链表结构*/
    unsigned long flags; /* REQ_ */

    sector_t sector; /* 要传送的下一个扇区 */
    unsigned long nr_sectors; /*要传送的扇区数目*/
    /*当前要传送的扇区数目*/
    unsigned int current_nr_sectors;

    sector_t hard_sector; /*要完成的下一个扇区*/
    unsigned long hard_nr_sectors; /*要被完成的扇区数目*/
    /*当前要被完成的扇区数目*/
    unsigned int hard_cur_sectors;

    struct bio *bio; /*请求的 bio 结构体的链表*/
    struct bio *biotail; /*请求的 bio 结构体的链表尾*/

    void *elevator_private;

    unsigned short ioprio;

    int rq_status;
    struct gendisk *rq_disk;
    int errors;
    unsigned long start_time;

    /*请求在物理内存中占据的不连续的段的数目,scatter/gather列表的尺寸*/
    unsigned short nr_phys_segments;

    /*与nr_phys_segments相同,但考虑了系统I/O MMU的remap */
    unsigned short nr_hw_segments;
```

```
    int tag;
    char *buffer; /*传送的缓冲,内核虚拟地址*/

    int ref_count; /* 引用计数 */
    ...
};
```

---

request结构体的主要成员包括：

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

上述 3个成员标识还未完成的扇区， hard_sector是第一个尚未传输的扇区， hard_nr_sectors是尚待完成的扇区数， hard_cur_sectors是当前 I/O操作中待完成的扇区数。这些成员只用于内核块设备层，驱动不应当使用它们，而应该与下面 3个成员打交道，如下所示：

```
sector_t sector;
unsigned long nr_sectors;
unsigned int current_nr_sectors;
```

这 3个成员在内核和驱动交互中发挥着重大作用。它们以 512字节大小为一个扇区，如果硬件的扇区大小不是 512字节，则需要进行相应的调整。例如，如果硬件的扇区大小是 2048字节，则在进行硬件操作之前，需要用 4来除起始扇区号。

hard_sector、 hard_nr_sectors、 hard_cur_sectors与 sector、nr_sectors、 current_nr_sectors之间可认为是“副本”关系。

```
struct bio*bio;
```

---

bio是这个请求中包含的 bio结构体的链表，驱动中不宜直接存取这个成员，而应该使用后文将介绍的 rq_for_each_bio（）。

```
char*buffer;
```

指向缓冲区的指针，数据应当被传送到或者来自这个缓冲区，这个指针是一个内核虚拟地址，可被驱动直接引用。

```
unsigned short nr_phys_segments;
```

该值表示相邻的页被合并后，这个请求在物理内存中占据的段的数目。如果设备支持分散 /聚集（ SG， scatter/gather）操作，可依据此字段申请 sizeof（ scatterlist） *nr_phys_segments的内存，并使用下列函数进行 DMA映射：

```
int blk_rq_map_sg（request_queue_t）*q，struct request*req，
struct scatterlist*sglist;
```

以上函数与 dma_map_sg（）类似，它返回 scatterlist列表入口的数量。

```
struct list_head queuelist;
```

用于链接这个请求到请求队列的链表结构，
blkdev_dequeue_request（）可用于从队列中移除请求。

使用如下宏可以从 request结构体获得数据传送的方向。返回值为 0表示从设备中读，返回值非 0表示向设备写。

```
rq_data_dir（struct request*req）;
```

2.请求队列

一个块请求队列是一个块 I/O请求的队列，其定义如代码清单 12-4。

代码清单 12-4　request队列结构体

```
1 struct request_queue
2 {
3     ...
4     /* 保护队列结构体的自旋锁 */
5     spinlock_t _ _queue_lock;
6     spinlock_t *queue_lock;
7
8     /* 队列kobject */
9     struct kobject kobj;
10
11     /* 队列设置 */
12     unsigned long nr_requests; /* 最大的请求数量 */
13     unsigned int nr_congestion_on;
14     unsigned int nr_congestion_off;
15     unsigned int nr_batching;
16
17     unsigned short max_sectors; /* 最大的扇区数 */
18     unsigned short max_hw_sectors;
19     unsigned short max_phys_segments; /* 最大的段数 */
20     unsigned short max_hw_segments;
21     unsigned short hardsect_size; /* 硬件扇区尺寸 */
22     unsigned int max_segment_size; /* 最大的段尺寸 */
23
24     unsigned long seg_boundary_mask; /* 段边界掩码 */
25     unsigned int dma_alignment; /* DMA 传送的内存对齐限制 */
26
27     struct blk_queue_tag *queue_tags;
28
29     atomic_t refcnt; /* 引用计数 */
30
31     unsigned int in_flight;
32
33     unsigned int sg_timeout;
```

```
34      unsigned int sg_reserved_size;
35      int node;
36
37      struct list_head drain_list;
38
39      struct request *flush_rq;
40      unsigned char ordered;
41  };
```

请求队列跟踪等候的块 I/O请求，它存储用于描述这个设备能够支持的请求类型信息、它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数，其结果是：如果请求队列被配置正确了，它不会交给该设备一个不能处理的请求。

请求队列还实现一个插入接口，这个接口允许使用多个 I/O调度器，I/O调度器（也称电梯）的工作是以最优性能的方式向驱动提交 I/O请求。大部分 I/O调度器累积批量的 I/O请求，并将它们排列为递增（或递减）的块索引顺序后提交给驱动。进行这些工作的原因在于，对于磁头而言，当给定顺序排列的请求时，可以使得磁盘顺序地从一头到另一头工作，非常像一个满载的电梯，在一个方向移动直到它的所有“请求”被满足。

另外， I/O调度器还负责合并邻近的请求，当一个新 I/O请求被提交给调度器后，它会在队列里搜寻包含邻近扇区的请求。如果找到一个，并且如果结果的请求不是太大，调度器将合并这两个请求。

对磁盘等块设备进行 I/O操作顺序的调度类似于电梯的原理，先服务完上楼的乘客再服务下楼的乘客，这样效率会更高，而顺序响应用户的请求则使电梯无序地忙乱。

Linux 2.6内核包含 4个 I/O调度器，它们分别是 Noop I/O scheduler、 Anticipatory I/O scheduler、 Deadline I/O scheduler与 CFQ I/O scheduler。

Noop I/O scheduler是一个简化的调度程序，它只做最基本的合并与排序。

Anticipatory I/O scheduler是当前内核中默认的 I/O调度器，它拥有非常好的性能，在 Linux 2.5内核中它就相当引人注意。在与

Linux 2.4内核进行的对比测试中，在 Linux 2.4内核中多项以分钟为单位完成的任务，它则是以秒为单位来完成的，正因为如此，它成为目前 Linux 2.6内核中默认的 I/O调度器。 Anticipatory I/O scheduler的缺点是比较庞大与复杂，在一些特殊的情况下，特别是在数据吞吐量非常大的数据库系统中它会变得比较缓慢。

Deadline I/O scheduler是针对 Anticipatory I/O scheduler的缺点进行改善而来的，表现出的性能几乎与 Anticipatory I/O scheduler一样好，但是比后者小巧。

CFQ I/O scheduler为系统内的所有任务分配相同的带宽，提供一个公平的工作环境，它比较适合桌面环境。事实上在测试中它也有不错的表现， mplayer、 xmms等多媒体播放器与它配合得相当好，回放平滑，几乎没有因访问磁盘而出现的跳帧现象。

内核 block目录中的 noop-iosched.c、 as-iosched.c、 deadline-iosched.c和 cfq-iosched.c文件分别实现了上述调度算法。

可以通过给 kernel添加启动参数，选择使用的 I/O调度算法，如：

```
kernel elevator=deadline
```

1）初始化请求队列。

```
request_queue_t*blk_init_queue（request_fn_proc*rfn,
spinlock_t*lock）;
```

上述函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的模块加载函数中调用。

2）清除请求队列。

```
void blk_cleanup_queue（request_queue_t*q）；
```

上述函数完成将请求队列返回给系统的任务，一般在块设备驱动模块卸载函数中调用。

而 blk_put_queue（）宏则定义为：

```
#define blk_put_queue（q）blk_cleanup_queue（（q））
```

3）分配“请求队列”。

```
request_queue_t*blk_alloc_queue（int gfp_mask）；
```

对于 Flash、 RAM盘等完全随机访问的非机械设备，并不需要进行复杂的 I/O调度，这个时候，应该使用上述函数分配一个“请求队列”，并使用如下函数来绑定请求队列和“制造请求”函数。

```
void blk_queue_make_request（request_queue_t*q，
make_request_fn*mfn）；
```

这种方式分配的“请求队列”实际上不包含任何请求，所以给其加上引号。

4）提取请求。

```
struct request*elv_next_request（request_queue_t*queue）；
```

上述函数用于返回下一个要处理的请求（由 I/O调度器决定），如果没有请求则返回 NULL。 elv_next_request（）不会清除请求，它仍然将这个请求保留在队列上，但是标识它为活动的，这个标识将阻止I/O调度器合并其他的请求到已开始执行的请求。因为elv_next_request（）不从队列里清除请求，因此连续调用它两次，两次会返回同一个请求结构体。

5）去除请求。

```
void blkdev_dequeue_request（struct request*req）；
```

上述函数从队列中去除一个请求。如果驱动中同时从同一个队列中操作了多个请求，它必须以这样的方式将它们从队列中去除。

如果需要将一个已经出列的请求归还到队列中，可以进行以下调用：

```
void elv_requeue_request（request_queue_t*queue，struct
request*req）；
```

另外，块设备层还提供了一套函数，这些函数可被驱动用来控制一个请求队列的操作，主要包括以下操作。

6）启停请求队列。

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果块设备到达不能处理等候的命令的状态，应调用blk_stop_queue（）来告知块设备层。之后，请求函数将不被调用，除非再次调用 blk_start_queue（）将设备恢复到可处理请求的状态。

7）参数设置。

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned
short max);
void blk_queue_max_phys_segments(request_queue_t *queue,
unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned
short max);
void blk_queue_max_segment_size(request_queue_t *queue,
unsigned int max);
```

上述函数用于设置描述块设备可处理的请求参数。
blk_queue_max_sectors（）描述任一请求可包含的最大扇区数，默认值为 255； blk_queue_max_phys_segments（）和 blk_queue_max_hw_segments（）都控制一个请求中可包含的最大物理段（系统内存中不相邻的区）， blk_queue_max_hw_segments（）考虑了系统 I/O内存管理单元的重映射，这两个参数默认都是 128。
blk_queue_max_segment_size告知内核请求段的最大字节数，默认值为 65536。

8）通告内核。

```
void blk_queue_bounce_limit（request_queue_t*queue,
u64dma_addr）;
```

该函数用于告知内核块设备执行 DMA时可使用的最高物理地址 dma_addr，如果一个请求包含超出这个限制的内存引用，系统将会给这个操作分配一个“反弹”缓冲区。这种方式的代价昂贵，因此应尽量避免使用。

可以给 dma_addr参数提供任何可能的值或使用预先定义的宏，如 BLK_BOUNCE_HIGH（对高端内存页使用反弹缓冲区），或者 BLK_BOUCE_ANY（驱动可在任何地址执行 DMA）等，默认值是 BLK_BOUNCE_HIGH。

```
blk_queue_segment_boundary（request_queue_t*queue，unsigned long mask）；
```

如果我们正在编写驱动的设备无法处理跨越一个特殊大小内存边界的请求，应该使用这个函数来告知内核这个边界。例如，如果设备处理跨 4MB边界的请求有困难，应该传递一个 0x3fffff掩码，默认的掩码是 0xffffffff（对应 4GB边界）。

```
void blk_queue_dma_alignment（request_queue_t*queue，int mask）；
```

告知内核块设备施加于 DMA传送的内存对齐限制，所有请求都匹配这个对齐，默认的屏蔽是 0x1ff，它导致所有的请求被对齐到 512字节边界。

```
void blk_queue_hardsect_size（request_queue_t*queue，unsigned short max）；
```

该函数告知内核块设备硬件扇区的大小，所有由内核产生的请求都是这个大小的倍数并且被正确对界。但是，内核块设备层和驱动之间的通信还是以 512字节扇区为单位进行。

3.块 I/O

通常一个 bio对应一个块 I/O请求，代码清单 12.5给出了 bio结构体的定义。 I/O调度算法可将连续的 bio合并成一个请求。所以，一个请求可以包含多个 bio。

代码清单 12-5　bio结构体

```
1 struct bio
```

```
2 {
3     sector_t bi_sector; /* 要传输的第一个扇区 */
4     struct bio *bi_next; /* 下一个bio */
5     struct block_device *bi_bdev;
6     unsigned long bi_flags; /* 状态、命令等 */
7     unsigned long bi_rw; /* 低位表示READ/WRITE,高位表示优先级*/
8
9     unsigned short bi_vcnt; /* bio_vec数量 */
10     unsigned short bi_idx; /* 当前bvl_vec索引 */
11
12     /*不相邻的物理段的数目*/
13     unsigned short bi_phys_segments;
14
15     /*物理合并和DMA remap合并后不相邻的物理段的数目*/
16     unsigned short bi_hw_segments;
17
18     unsigned int bi_size; /* 以字节为单位所需传输的数据大小 */
19
20     /* 为了明了最大的hw尺寸,我们考虑这个bio中第一个和最后一个
21 虚拟的可合并的段的尺寸 */
22     unsigned int bi_hw_front_size;
23     unsigned int bi_hw_back_size;
24
25     unsigned int bi_max_vecs; /* 我们能持有的最大bvl_vecs数 */
26
27     struct bio_vec *bi_io_vec; /* 实际的vec列表 */
28
29     bio_end_io_t *bi_end_io;
30     atomic_t bi_cnt;
31
32     void *bi_private;
33
34     bio_destructor_t *bi_destructor; /* destructor */
35 };
```

下面我们对其中的核心成员进行分析：

```
sector_t bi_sector;
```

标识这个 bio结构体要传送的第一个（ 512字节）扇区。

```
unsigned int bi_size;
```

被传送的数据大小，以字节为单位，驱动中可以使用 bio_sectors（ bio）宏获得以扇区为单位的大小。

```
unsigned long bi_flags;
```

一组描述 bio结构体的标志，如果这是一个写请求，最低有效位被置位，可以使用 bio_data_dir（ bio）宏来获得读写方向。

```
unsigned short bio_phys_segments;
unsigned short bio_hw_segments;
```

分别表示包含在这个 bio结构体中要处理的不连续的物理内存段的数目和考虑 DMA重映像后的不连续的内存段的数目。

bio结构体的核心是一个称为 bi_io_vec的数组，它由 bio_vec结构体组成， bio_vec结构体的定义如代码清单 12-6所示。

代码清单 12-6　bio_vec结构体

```
struct bio_vec
{
    struct page *bv_page; /* 页指针 */
    unsigned int bv_len; /* 传输的字节数 */
    unsigned int bv_offset; /* 偏移位置 */
};
```

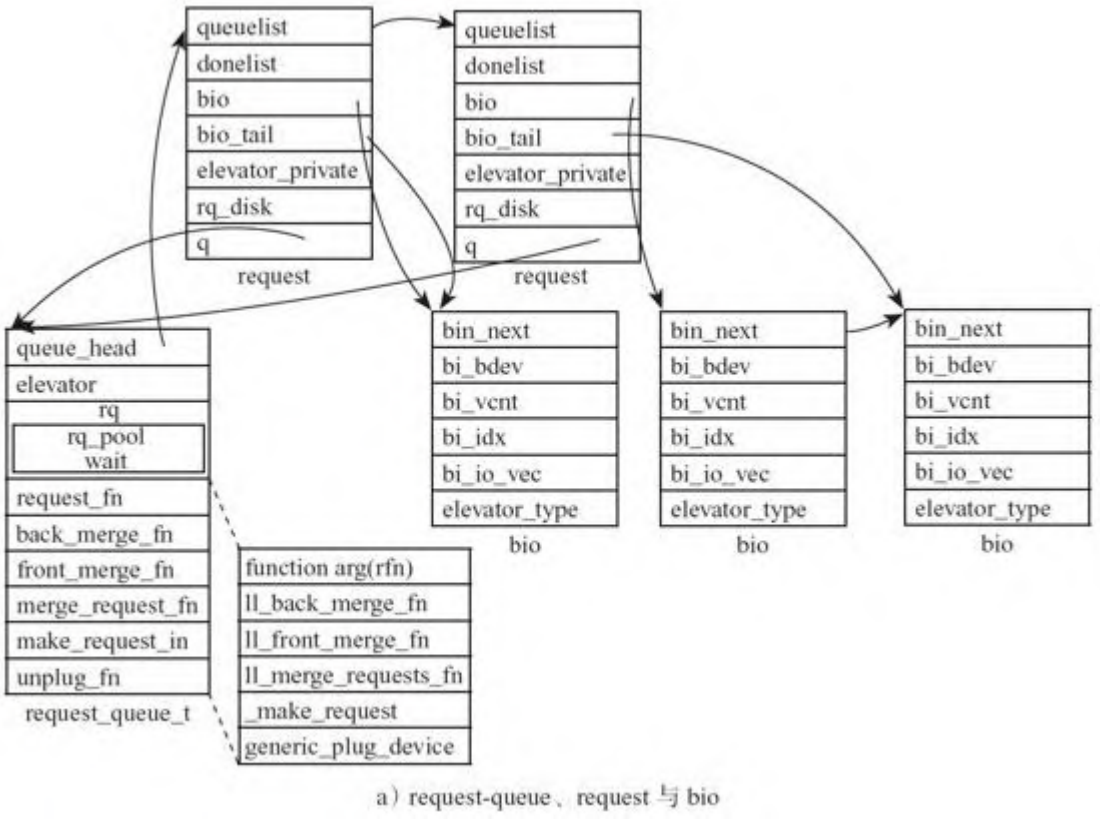
我们不应该直接访问 bio结构体的 bio_vec成员，而应该使用 bio_for_each_segment（）宏来进行这项工作，可以用这个宏循环遍

历整个 bio结构体中的每个段，这个宏的定义如代码清单 12-7所示。

代码清单 12-7　bio_for_each_segment（）宏

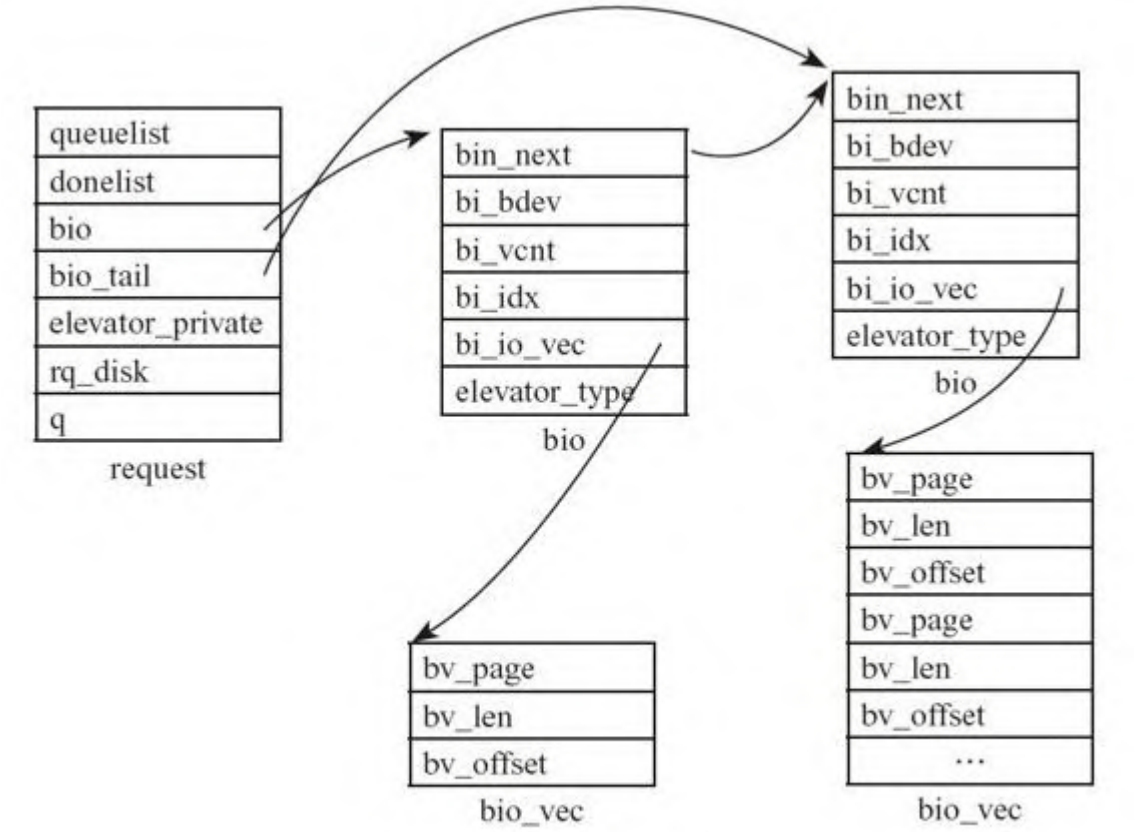
```
#define _ _bio_for_each_segment(bvl, bio, i, start_idx) \
for (bvl = bio_iovec_idx((bio), (start_idx)), i = (start_idx);
\
i < (bio)->bi_vcnt; \
bvl++, i++)

#define bio_for_each_segment(bvl, bio, i) \
_ _bio_for_each_segment(bvl, bio, i, (bio)->bi_idx
```

图 12-2a所示为 request队列、 request与 bio数据结构之间的关系，图 12-2b所示为 request、 bio和 bio_vec数据结构之间的关系，图 12-2c所示为 bio与 bio_vec数据结构之间的关系，因此整个图 12-2递归地呈现了 request队列、 request、 bio和 bio_vec这 4个结构体之间的关系。



a）request-queue、request 与 bio

图12-2 request队列、 request、 bio与 bio_vec 4个结构体关系



b）request、bio 与 bio_vec

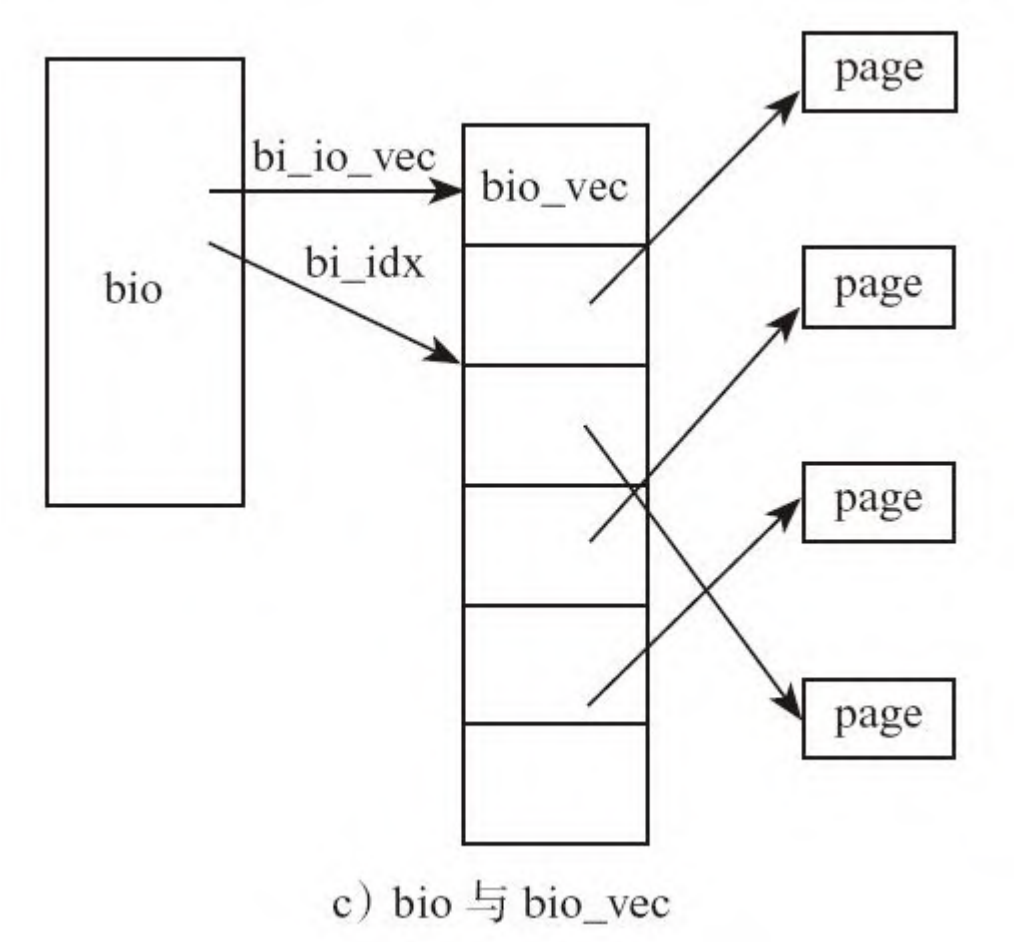


图 12-2 （续）

内核还提供了一组函数（宏）用于操作 bio：

```
int bio_data_dir（struct bio*bio）；
```

上述函数可用于获得数据传输的方向是 READ还是 WRITE。

```
struct page*bio_page（struct bio*bio）；
```

上述函数可用于获得目前的页指针。

```
int bio_offset（struct bio*bio）；
```

上述函数返回操作对应的当前页内的偏移，通常块 I/O操作本身就是页对齐的。

```
int bio_cur_sectors（struct bio*bio）；
```

上述函数返回当前 bio_vec要传输的扇区数。

```
char*bio_data（struct bio*bio）；
```

上述函数返回数据缓冲区的内核虚拟地址。

```
char*bvec_kmap_irq（struct bio_vec*bvec，unsigned long*flags）；
```

上述函数返回一个内核虚拟地址，这个地址可用于存取被给定的bio_vec入口指向的数据缓冲区。它也会屏蔽中断并返回一个原子kmap，因此，在 bvec_kunmap_irq（）被调用以前，驱动不应该睡眠。

```
void bvec_kunmap_irq（char*buffer，unsigned long*flags）；
```

上述函数是 bvec_kmap_irq（）函数的“反函数”，它撤销bvec_kmap_irq（）创建的映射。

```
char*bio_kmap_irq（struct bio*bio，unsigned long*flags）；
```

上述函数是对 bvec_kmap_irq（）的包装，它返回给定的 bio的当前 bio_vec入口的映射。

```
char*__bio_kmap_atomic（struct bio*bio，int i，enum km_type
type）；
```

上述函数通过 kmap_atomic（）获得返回给定 bio的第 i个缓冲区的虚拟地址。

```
void__bio_kunmap_atomic（char*addr，enum km_type type）；
```

上述函数返回由 __bio_kmap_atomic（）获得的内核虚拟地址。

另外，对 bio的引用计数通过如下函数完成：

```
void bio_get(struct bio *bio); //引用bio
void bio_put(struct bio *bio); //释放对bio的引用
```

12.2.4 块设备驱动注册与注销

块设备驱动中的第一个工作通常是注册它们自己到内核，完成这个任务的函数是 register_blkdev（），其原型为：

```
int register_blkdev（unsigned int major，const char*name）；
```

major参数是块设备要使用的主设备号， name为设备名，它会在/proc/devices中被显示。如果 major为 0，内核会自动分配一个新的主设备号， register_blkdev（）函数的返回值就是这个主设备号。如果 register_blkdev（）返回一个负值，表明发生了一个错误。

与 register_blkdev（）对应的注销函数是 unregister_blkdev（），其原型为：

```
int unregister_blkdev(unsigned int major, const char*name);
```

这里，传递给 unregister_blkdev（）的参数必须与传递给 register_blkdev（）的参数匹配，否则这个函数返回 -EINVAL。

值得一提的是，在 Linux 2.6内核中，对 register_blkdev（）的调用完全是可选的， register_blkdev（）的功能正在减少，这个调用最多只完成两件事：

1）如果需要，分配一个动态主设备号。

2）在 /proc/devices中创建一个入口。

在将来的内核中， register_blkdev（）可能会被删除。但是目前大部分驱动仍然调用它。代码清单 12-8给出了一个块设备驱动注册的模板。

代码清单 12-8　块设备驱动注册模板

```
xxx_major = register_blkdev(xxx_major, "xxx");
if (xxx_major <= 0) //注册失败
{
    printk(KERN_WARNING "xxx: unable to get major number\n");
    return -EBUSY;
}
```

12.3　Linux块设备驱动的模块加载与卸载

在块设备驱动的模块加载函数中通常需要完成如下工作：

1）分配、初始化请求队列，绑定请求队列和请求函数。

2）分配、初始化 gendisk，给 gendisk的 major、 fops、 queue等成员赋值，最后添加 gendisk。

3）注册块设备驱动。

代码清单 12-9和 12-10分别给出了使用 blk_alloc_queue（）分配请求队列并使用 blk_queue_make_request（）绑定“请求队列”和“制造请求”的函数，以及使用 blk_init_queue（）初始化请求队列并绑定请求队列与请求处理函数这两种不同情况下的块设备驱动模块加载函数模板。

代码清单 12-9　块设备驱动的模块加载函数模板（使用 blk_alloc_queue）

```
static int _ _init xxx_init(void)
{
    //分配gendisk
    xxx_disks = alloc_disk(1);
    if (!xxx_disks)
    {
        goto out;
    }

    //块设备驱动注册
    if (register_blkdev(XXX_MAJOR, "xxx"))
    {
        err = - EIO;
        goto out;
    }

    //"请求队列"分配
    xxx_queue = blk_alloc_queue(GFP_KERNEL);
    if (!xxx_queue)
```

```
    {
        goto out_queue;
    }

    blk_queue_make_request(xxx_queue, &xxx_make_request); //绑定"制造请求"函数
    blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇区尺寸设置

    //gendisk初始化
    xxx_disks->major = XXX_MAJOR;
    xxx_disks->first_minor = 0;
    xxx_disks->fops = &xxx_op;
    xxx_disks->queue = xxx_queue;
    sprintf(xxx_disks->disk_name, ?xxx%d", i);
    set_capacity(xxx_disks, xxx_size); //xxx_size以512字节为单位
    add_disk(xxx_disks); //添加gendisk

    return 0;
out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
out: put_disk(xxx_disks);
    blk_cleanup_queue(xxx_queue);

    return - ENOMEM;
}
```

代码清单 12-10　块设备驱动的模块加载函数模板（使用 blk_init_queue）

```
static int _ _init xxx_init(void)
{
    //块设备驱动注册
    if (register_blkdev(XXX_MAJOR, "xxx"))
    {
        err = - EIO;
        goto out;
    }

    //请求队列初始化
    xxx_queue = blk_init_queue(xxx_request, xxx_lock);
    if (!xxx_queue)
    {
        goto out_queue;
```

```
    }

    blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇
区尺寸设置

    //gendisk初始化
    xxx_disks->major = XXX_MAJOR;
    xxx_disks->first_minor = 0;
    xxx_disks->fops = &xxx_op;
    xxx_disks->queue = xxx_queue;
    sprintf(xxx_disks->disk_name, ?xxx%d", i);
    set_capacity(xxx_disks, xxx_size *2);
    add_disk(xxx_disks); //添加gendisk

    return 0;
out_queue: unregister_blkdev(XXX_MAJOR, ?xxx");
out: put_disk(xxx_disks);
    blk_cleanup_queue(xxx_queue);

    return - ENOMEM;
}
```

在块设备驱动的模块卸载函数中完成与模块加载函数相反的工作。

1）清除请求队列。

2）删除 gendisk和对 gendisk的引用。

3）删除对块设备的引用，注销块设备驱动。

代码清单 12-11给出了块设备驱动的模块卸载函数的模板。

代码清单 12-11　块设备驱动的模块卸载函数模板

```
static void _ _exit xxx_exit(void)
{
    if (bdev)
    {
        invalidate_bdev(xxx_bdev, 1);
        blkdev_put(xxx_bdev);
    }
```

```
    del_gendisk(xxx_disks); //删除gendisk
    put_disk(xxx_disks);
    blk_cleanup_queue(xxx_queue[i]); //清除请求队列
    unregister_blkdev(XXX_MAJOR, "xxx");
}
```

12.4 块设备的打开 /释放 /IOCTL

块设备驱动的 open（）和 release（）函数并非是必需的，一个简单的块设备驱动可以不提供 open（）和 release（）函数。

块设备驱动的 open（）函数与其字符设备驱动的对等体非常类似，都以相关的 inode和 file结构体指针作为参数。当一个节点引用一个块设备时， inode->i_bdev->bd_disk包含一个指向关联 gendisk结构体的指针。因此，类似于字符设备驱动，我们也可以将 gendisk的 private_data赋给 file的 private_data， private_data同样最好是指向描述该设备的设备结构体 xxx_dev的指针，如代码清单 12-12所示。

代码清单 12-12 在块设备的 open（）函数中赋值 private_data

```
static int xxx_open(struct inode *inode, struct file *filp)
{
    struct xxx_dev *dev = inode->i_bdev->bd_disk->private_data;
    filp->private_data = dev; //赋值file的private_data
    ...
    return 0;
}
```

在一个处理真实硬件设备的驱动中， open（）和 release（）方法还应当设置驱动和硬件的状态，这些工作可能包括启停磁盘、加锁一个可移出设备和分配 DMA缓冲等。

与字符设备驱动一样，块设备可以包含一个 ioctl（）函数以提供对设备的 I/O控制能力。实际上，高层的块设备层代码处理了绝大多数 ioctl（），因此，具体的块设备驱动中通常不再需要实现很多 ioctl命令。

代码清单 12-13给出的 ioctl（）函数只实现一个命令 HDIO_GETGEO，用于获得磁盘的几何信息（ geometry，指 CHS，即 Cylinder、 Head、 Sector/Track）

代码清单 12-13　块设备驱动的 I/O控制函数模板

```
int xxx_ioctl(struct inode *inode, struct file *filp, unsigned
int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo;
    struct xxx_dev *dev = filp->private_data; //通过file-
>private 获得设备结构体

    switch (cmd)
    {
        case HDIO_GETGEO:
            size = dev->size *(hardsect_size /
KERNEL_SECTOR_SIZE);
            geo.cylinders = (size &~0x3f) >> 6;
            geo.heads = 4;
            geo.sectors = 16;
            geo.start = 4;
            if (copy_to_user((void _ _user*)arg, &geo,
sizeof(geo)))
            {
                return - EFAULT;
            }
            return 0;
    }

    return - ENOTTY; //不知道的命令
}
```

第13章　Linux网络设备驱动

Linux系统中有三大类，前面我们已经讲过了字符类和块类，还有一类就是网络类。网络设备驱动比较特殊，它不再秉承“一切皆文件”的思想，也就是说，上层程序与网络设备的交互不再基于文件了。

13.1　Linux网络设备驱动体系结构

Linux网络设备驱动程序是 Linux网络子系统的一部分，对 ISO网络七层协议来讲，它位于 TCP/IP网络体系结构的网络接口层，主要实现上层协议栈与网络设备的数据交换。

如图 13-1所示， Linux网络设备驱动程序的体系结构可划分为 4层。

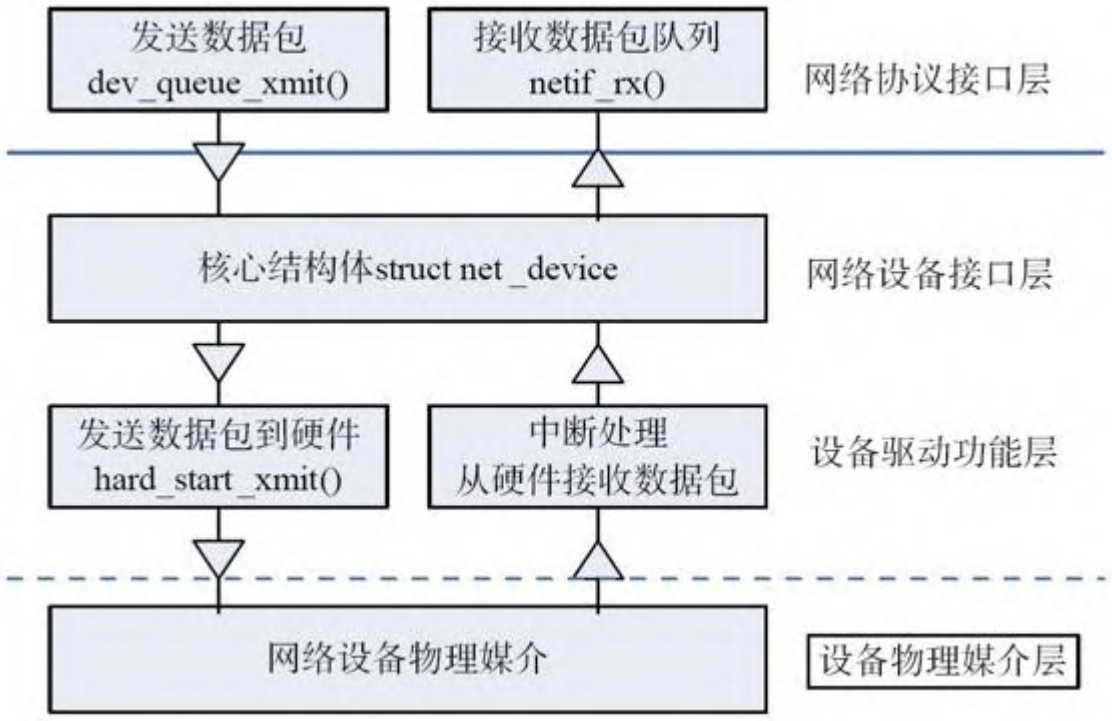


图13-1　Linux网络设备驱动体系结构图

Linux中所有的网络设备都抽象为一个统一的接口，即网络设备接口，通过 struct net_device类型的结构体变量表示网络设备在内核中的运行情况，这里既包括回环（ loopback）设备，也包括硬件网络设备接口。内核通过以 dev_base为头指针的设备链表来管理所有的网络设备。

Linux的网络系统主要是基于 BSD UNIX的套接字（ socket）机制，与字符设备和块设备不同，网络设备没有对应地映射到文件系统中的设

备节点。也就是讲，上层应用是通过 socket套接字，经过网络协议接口层来使用具体的网络设备驱动，以及访问实际的网络设备。

13.2 Linux网络设备驱动结构

如上节所述， Linux使用 struct net_device结构体来代表网络设备。定义如代码清单 13-1所示。

代码清单 13-1 net_device结构体

```
struct net_device {
    char                    name[IFNAMSIZ];        //网络设备名
    ...

    struct hlist_node       name_hlist;              //网络设备名列表
    ...

    unsigned long                   mem_end;           //网络发送或接收内存结束地址
    unsigned long                   mem_start;              //网络发送或接收内存起始地址
    unsigned long                   base_addr;              //网络设备 I/O 基地址
    unsigned int              irq;                    //网络设备 IRQ 号
    ...
    unsigned char              dma;                     //DMA 通道

    ...

    struct list_head       dev_list;              //网络设备列表
    ...
    unsigned int              mtu;                     //最大传输单元
    ...
    unsigned char              *dev_addr;              //MAC地址

    ...
};
```

事实上，在结构体 struct net_device中，还定义了以下的功能函数：

1） int（ *init）（ struct net_device*dev）：设备初始化和向系统注册的函数，仅调用一次。

2） int（ *open）（ struct net_device*dev）：设备打开接口函数，当用 ifconfig激活网络设备时被调用，注册所用的系统资源（I/O端口、 IRQ、 DMA等）同时激活硬件并增加使用计数。

3） int（ *stop）（ struct net_device*dev）：执行 open方法的反操作。

4） *hard_start_xmit：初始化数据包传输的函数。

5） *hard_header：该函数（在 hard_start_xmit前被调用）根据先前检索到的源和目标硬件地址建立硬件头。

13.3　Linux网络设备驱动 I/O实现

Linux内核源代码中提供了网络设备接口及上层网络协议接口的代码，因此移植特定网络硬件的驱动程序的主要工作就是完成设备驱动功能层的相应代码，根据底层具体的硬件特性，定义网络设备接口 struct net_device类型的结构体变量，并实现其中相应的操作函数及中断处理程序。

Linux网络系统各个层次之间的数据传送都是通过套接字缓冲区 sk_buff完成的， sk_buff数据结构是各层协议数据处理的对象，sk_buff也是驱动程序与网络之间交换数据的媒介。驱动程序向网络发送数据时，必须从其中获取数据源和数据长度；驱动程序从网络上接收到数据后也要将数据保存到 sk_buff中才能交给上层协议处理。

13.3.1　网络设备初始化

网络设备驱动在 Linux内核中是以内核模块的形式存在的，对应于模块的初始化，需要提供一个初始化函数来初始化网络设备的硬件寄存器、配置 DMA以及初始化相关内核变量等。设备初始化函数在内核模块被加载时调用，它的函数形式如代码清单 13-2所示。

代码清单 13-2　net_device结构体

```
static int __init xx_init (void) {
    ...
}
module_init(xx_init);   //表明模块加载时自动调用 xx_init 函数
```

设备初始化函数主要完成以下功能：

1.硬件初始化

因为网络设备主要分为 PHY、 MAC和 DMA三个硬件模块，开发者需要分别对这三个模块进行初始化。

1）初始化 PHY模块，包括设置双工 /半双工运行模式、设备运行速率和自协商模式等。

2）初始化 MAC模块，包括设置设备接口模式等。

3）初始化 DMA模块，包括建立 BD表、设置 BD属性以及给 BD分配缓存等。

2.内核变量初始化

初始化并注册内核设备，实际上就是初始化 net_device结构体中的属性变量。开发者需要申请该变量对应的空间（通过 alloc_netdev函数）、设置变量参数、挂接接口函数以及注册设备（通过 register_netdev函数）。

常用的挂接接口函数如代码清单 13-3所示。

代码清单 13-3　net_device内核变量初始化

```
net_device *dev_p;
dev_p->open               = xx_open;        // 设备打开函数
dev_p->stop               = xx_stop;        // 设备停止函数
dev_p->hard_start_xmit        = xx_tx;        // 数据发送函数
dev_p->do_ioctl               = xx_ioctl;        // 其他控制函数
...
```

13.3.2　网络数据包的收发

数据的接收和发送是网络设备驱动最重要的部分，对于用户来说，他们无须了解当前系统使用了什么网络设备、网络设备收发如何进行等，所有的这些细节对于用户都是屏蔽的。 Linux使用 socket作为连接用户和网络设备的一个桥梁。用户可以通过 read（） /write（）等函数操作 socket，然后通过 socket与具体的网络设备进行交互，从而进行实际的数据收发工作。

用户传给 socket的数据首先会保存在 sk_buff对应的缓冲区中，sk_buff的结构定义在 include/linux/skbuff.h文件中。它保存数据

包的结构示意图如图 13-2所示。

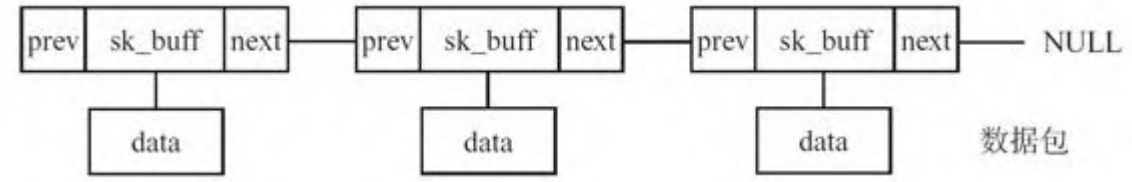


图13-2　sk_buff数据结构图

1.数据发送流程

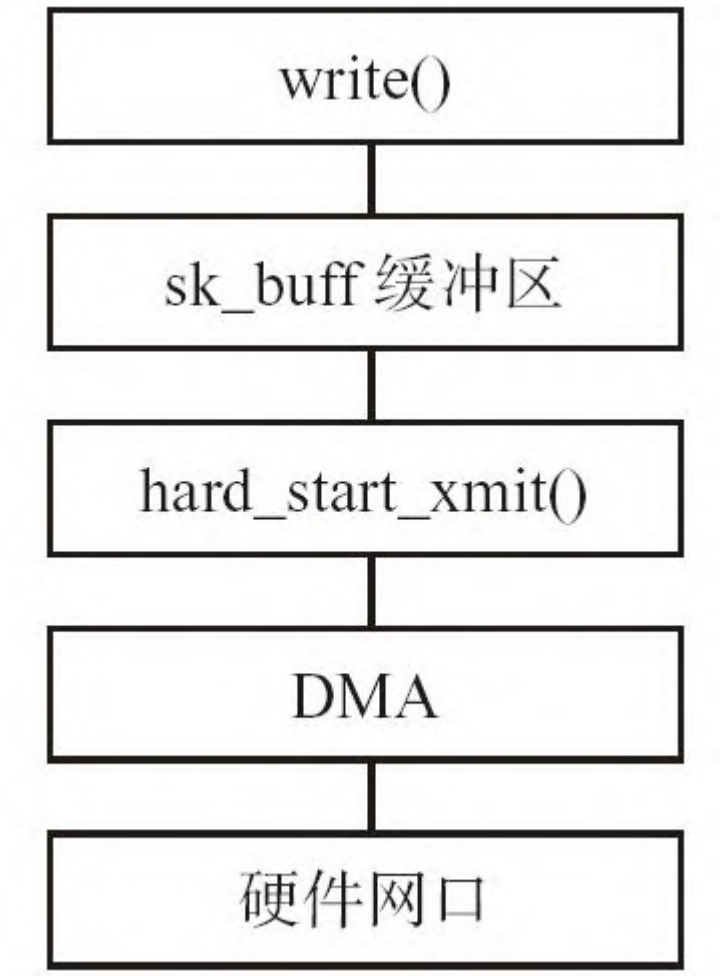


图 13-3　数据发送流程图

当用户调用 socket开始发送数据时，数据被存储到了 sk_buff类型的缓存中，网络设备的发送函数（设备初始化函数中注册的 hard_start_xmit）也随之被调用，流程图如图 13-3所示。

1）用户首先创建一个 socket，然后调用 write（）之类的写函数通过 socket访问网络设备，同时将数据保存在 sk_buff类型的缓冲区中。

2） socket接口调用网络设备发送函数（ hard_start_xmit（））， hard_start_xmit（）已经在初始化过程中被挂接成类似于 xx_tx的具体发送函数， xx_tx主要实现如下步骤。

①从发送 BD表中取出一个空闲的 BD。

②根据 sk_buff中保存的数据修改 BD的属性，一个是数据长度，另一个是数据包缓存指针。值得注意的是，数据包缓存指针对应的必须是物理地址，这是因为 DMA在获取 BD中对应的数据时只能识别存储该数据缓存的物理地址。

```
bd_p->length = skb_p->len;
bd_p->bufptr = virt_to_phys(skb_p->data);
```

③修改该 BD的状态为就绪态， DMA模块将自动发送处于就绪态 BD中所对应的数据。

④移动发送 BD表的指针指向下一个 BD。

3） DMA模块开始将处于就绪态 BD缓存内的数据发送至网络中，当发送完成后自动恢复该 BD为空闲态。

2.数据接收流程

当网络设备接收到数据时， DMA模块会自动将数据保存起来并通知处理器来取，处理器通过中断或者轮询方式发现有数据接收进来后，再将数据保存到 sk_buff缓冲区中，并通过 socket接口读出来。流程图如图 13-4所示。

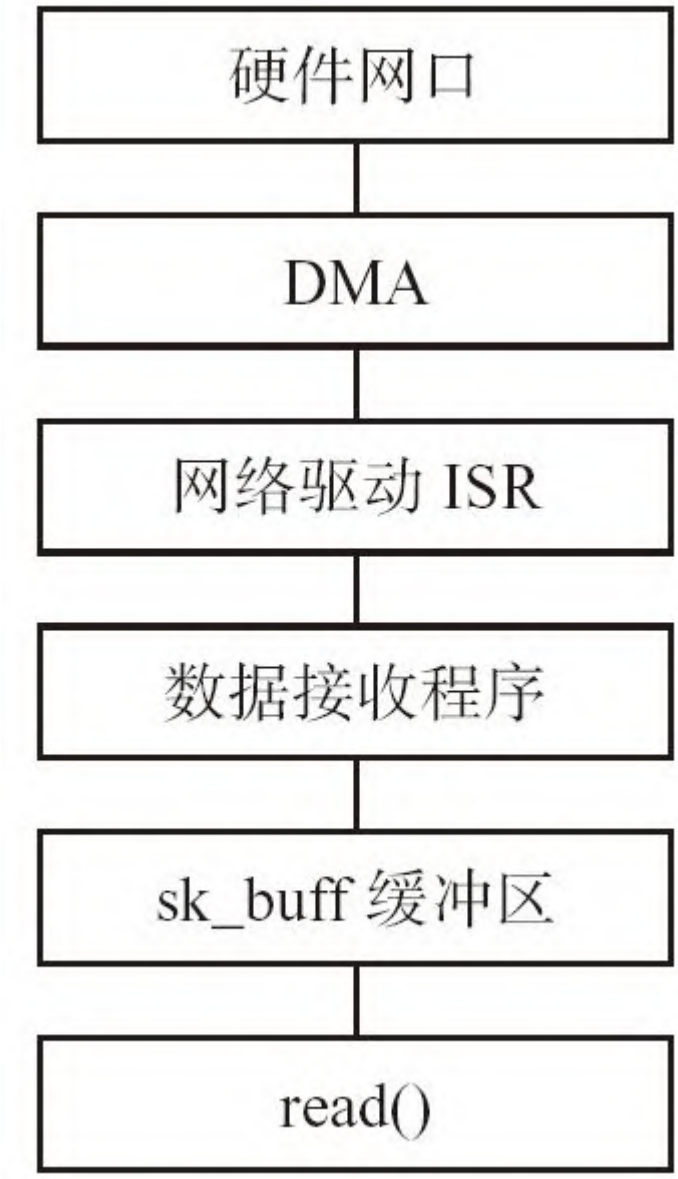


图 13-4　数据接收流程图

1）网络设备接收到数据后， DMA模块搜索接收 BD表，取出空闲的 BD，并将数据自动保存到该 BD的缓存中，修改 BD为就绪态，并同时触发中断（该步骤可选）。

2）处理器可以通过中断或者轮询的方式检查接收 BD表的状态，无论采用哪种方式，它们都需要实现以下步骤。

①从接收 BD表中取出一个空闲的 BD。

②如果当前 BD为就绪态，检查当前 BD的数据状态，更新数据接收统计。

③从 BD中取出数据保存在 sk_buff缓冲区中。

④更新 BD的状态为空闲态。

⑤移动接收 BD表的指针指向下一个 BD。

3）用户调用 read（）之类的读函数，从 sk_buff缓冲区中读出数据，同时释放该缓冲区。

Linux内核在接收数据时有两种方式可供选择，一种是中断方式，另外一种是轮询方式。

如果选择中断方式，首先在使用该驱动之前，需要注册该中断对应的中断类型号和中断处理程序。网络设备驱动在初始化时会将具体的xx_open函数挂接在驱动的 open接口上， xx_open函数挂接中断如下：

```
request_irq(rx_irq, xx_isr_rx, ... );
request_irq(tx_irq, xx_isr_tx, ... );
```

网络设备的中断一般会分为两种，一种是发送中断，另一种是接收中断。内核需要分别对这两种中断类型号进行注册。

响应发送中断的发送中断处理程序（ xx_isr_tx）需要监控数据发送的情况，完成数据发送统计值的更新，驱动网络设备发送数据等；响应接收中断的接收中断处理程序（ xx_isr_rx）则需要接收数据，将接收到的数据转送给上层协议层，同时监控数据接收状态，完成数据接收统计值的更新等。

对于中断方式来说，由于每接收到一个包都会产生一个中断，而处理器会迅速跳到中断服务程序中去处理，因此中断接收方式的实时性高，但如果遇到数据包流量很大的情况时，过多的中断会增加系统的负荷。

如果采用轮询方式，就不需要使能网络设备的中断状态，也不需要注册中断处理程序。操作系统会专门开启一个任务去定时检查 BD表，如果发现当前指针指向的 BD非空闲，则将该 BD对应的数据取出来，并恢复 BD的空闲状态。

由于是采用任务定时检查的原理，从而轮询接收方式的实时性较差，但它没有中断的系统上下文切换的开销，因此轮询方式在处理大流量数据包时会显得更加高效。

第三篇　实践出真知——Android驱动实践篇

经过前面艰苦而又有趣的学习，我们已拥有一定的Linux驱动开发基础，并对Android的架构有一定的了解。下面我们将开始真正的Android驱动开发之旅。

这里，我们将与一般介绍Android驱动开发的书籍不同，不以具体驱动数量取胜，而是以几个典型例子让读者对Android的驱动工作原理，以及Android上层与底层具体的交互有一个更通透的理解。

第14章　Android HAL层的设计

我们不急于开始Android具体驱动的讲解，而是从Android HAL入手来看看Linux驱动是如何与Android结合为一体的。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

14.1 Android HAL概述

从图 2-1Android的 4层架构中，在 C++库层的最底部，有一个子层——硬件抽象层（ Hardware Abstract Layer， HAL）。该层实现 Android上层对 Linux中驱动程序的调用，向 Android上层提供访问底层设备的接口，以便 Android系统的其他部分访问这些硬件。

Android驱动与一般 Linux驱动的最大区别就是把对硬件的支持分成了两层：一层放在了用户空间，我们称之为硬件抽象层；另一层则放到了内核空间，我们称之为内核驱动层。其中，一般内核驱动层只提供简单的访问硬件的逻辑，例如提供读写硬件寄存器方法，而具体读写什么值到硬件的逻辑，则放到硬件抽象层中。

通过这种分层的方式，硬件厂商不用再提供其 HAL层的源码，进而有利于保护这些硬件厂商的知识产权。另外通过这种分层的方式，有利于屏蔽底层驱动的差异性，为同一类设备向上提供统一的接口。

另外，实现 HAL有两种不同的方式，如图 14-1所示。其中，libhardware_legacy是老式的 HAL结构，各驱动逻辑在这里直接实现为一个个 .so动态链接库，上层应用或 framework通过这些 .so库，实现底层硬件访问；而 libhardware是新式 HAL结构，采用 Stub代理调用方式，由 Stub向 HAL提供 operations方法，进而向上层提供对底层硬件的操作。事实上，新式 HAL结构中的 Stub仍是以 *.so的形式存在，但 HAL已经将 *.so的具体实现方式隐藏起来了；也就是说对上层程序开发人员而言，不再需要它来确保某个驱动的 .so库已被装载与运行，只需要调用 libandroid_runtime相关的接口函数即可。runtime会负责符合 Android新 HAL结构 Stub驱动的加载。而且 runtime通过 Stub提供的 .so获取它的 operations方法，并告知 runtime的 callback方法，因此 runtime与 Stub都有可供对方调用的方法：一个应用请求通过 runtime调用 Stub的 operations方法，Stub在完成 operations方法响应后，再调用 runtime的 callback方法进行返回。

同时，细心的读者会从图 14-1中看到 3个词： AIDL、 JNI、 HAL。它们拥有共同的特性：它们是衔接两层之间的中间件技术。

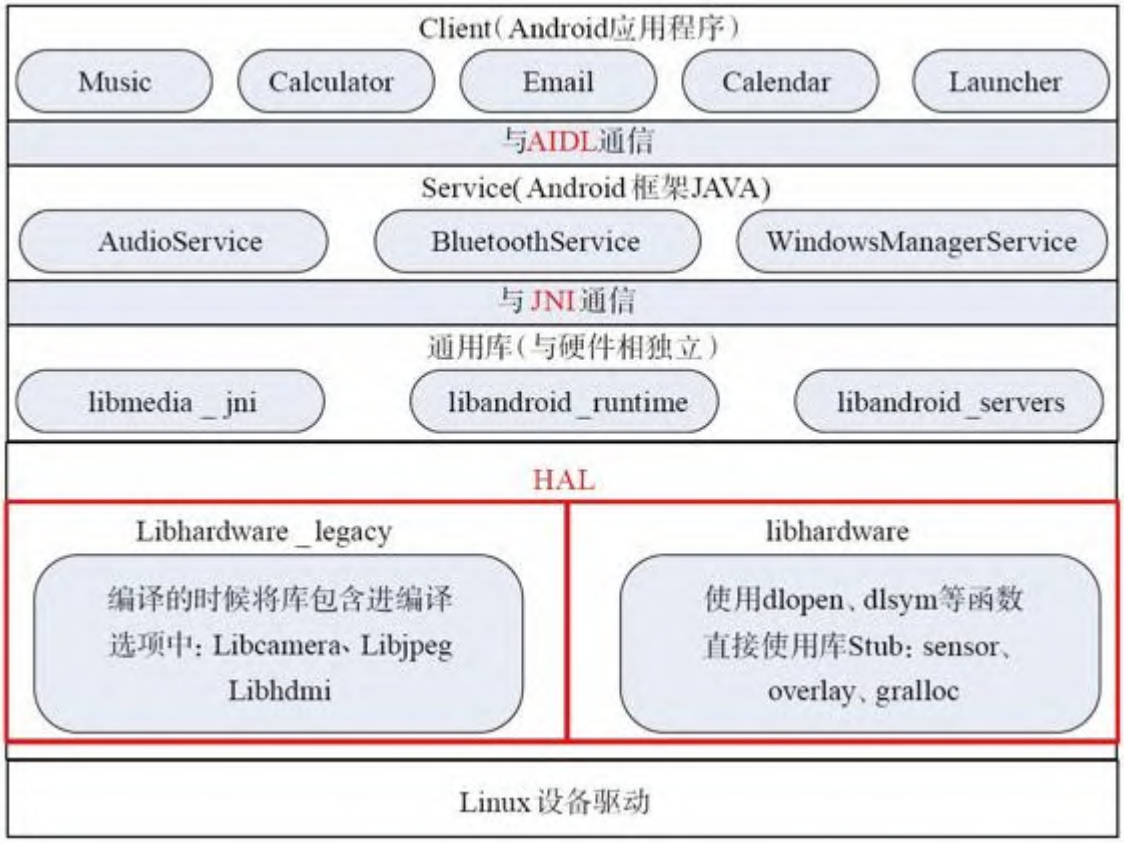


图14-1　实现HAL的两种结构

14.2 为 Android开发虚拟驱动 virtualio

为了方便描述为 Android系统编写与调用内核驱动程序的过程，我们假设一个虚拟的硬件设备，这个设备有 4个字节寄存器，它可读可写。同时，将该虚拟设备的内核驱动程序命名为 virtualio驱动程序。其实， Android内核驱动程序与一般 Linux内核驱动程序的编写方法一样，都是以 Linux模块的形式实现的。

不过，这里我们先从 Android系统的角度来描述 Android内核驱动程序的编写和编译过程。

1） 2.3节，完成 Android SDK源码载、开发环境搭建，其中包括了 Android内核驱动环境的搭建。

2）进入 kernel/drivers目录，新建 virtualio目录：

```
$ cd kernel/drivers
$ mkdir virtualio
```

3）在 virtualio目录中增加 virtualio.h文件。参见代码清单 14-1。

代码清单 14-1 virtualio.h

```
1.    #ifndef?_VIRTUALIO_ANDROID_H_ ??
2.    #define?_VIRTUALIO_ANDROID_H_ ??
3.    ??
4.    #include?<linux/cdev.h> ??
5.    #include?<linux/semaphore.h> ??
6.    ??
7.    #define?VIRTUALIO_DEVICE_NODE_NAME??"virtualio" ??
8.    #define?VIRTUALIO_DEVICE_FILE_NAME??"virtualio" ??
9.    #define?VIRTUALIO_DEVICE_PROC_NAME??"virtualio" ??
10.    #define?VIRTUALIO_DEVICE_CLASS_NAME?"virtualio" ??
11.    ??
12.    struct?virtualio_android_dev?{??
13.    char reg1;
14.    char reg2;
15.    char reg3;
16.    char reg4;
17.    struct?semaphore?sem;??
```

```
18.    struct?cdev?dev;??
19.    };??
20.    ??
21.    #endif??
```

其中第 7～ 10行定义了一些字符串常量宏，在后面我们要用到。此外，还定义了一个字符设备结构体 virtualio_android_dev，这个就是我们虚拟的硬件设备了；第 13~16行代码中的成员变量就代表设备里面的寄存器，它们的类型为 char， sem成员变量是一个信号量，以同步访问寄存器， dev成员变量是一个内嵌的字符设备。

4）在 virtualio目录中增加 virtualio.c文件，这是该驱动程序的实现部分。驱动程序的功能主要是向上层提供访问设备的寄存器的值，包括读和写。这里提供了三种访问设备寄存器的方法，一是通过 proc文件系统来访问，二是通过传统的设备文件的方法来访问，三是通过 devfs文件系统来访问。参见代码清单 14-2~14-5。

代码清单 14-2　virtualio.c——驱动骨架部分

```
1.    #include?<linux/init.h> ??
2.    #include?<linux/module.h> ??
3.    #include?<linux/types.h> ??
4.    #include?<linux/fs.h> ??
5.    #include?<linux/proc_fs.h> ??
6.    #include?<linux/device.h> ??
7.    #include?<asm/uaccess.h> ??
8.    ??
9.    #include?"virtualio.h" ??
10.    ??
11.    /*主设备和从设备号变量*/??
12.    static?int?virtualio_major?=?0;??
13.    static?int?virtualio_minor?=?0;??
14.    ??
15.    /*设备类别和设备变量*/??
16.    static?struct?class*?virtualio_class?=?NULL;??
17.    static?struct?virtualio_android_dev*?virtualio_dev?=?NULL;??
18.    ??
19.    /*传统的设备文件操作方法*/??
20.    static?int?virtualio_open(struct?inode*?inode,?struct?file*?
filp);??
21.    static?int?virtualio_release(struct?inode*?inode,?struct?file*?
filp);??
22.    static?ssize_t?virtualio_read(struct?file*?filp,?char?__user?
*buf,u8 num);??
23.    static?ssize_t?virtualio_write(struct?file*?filp,?const?char?
```

```
__user *buf,u8 num);??
24.    ??
25.    /*设备文件操作方法表*/??
26.    static?struct?file_operations?virtualio_fops?=?{??
27.    ????.owner?=?THIS_MODULE,??
28.    ????.open?=?virtualio_open,??
29.    ????.release?=?virtualio_release,??
30.    ????.read?=?virtualio_read,??
31.    ????.write?=?virtualio_write,???
32.    };??
33.    ??
34.    /*访问设置属性方法*/??
35.    static?ssize_t?virtualio_reg_show(struct?device*?dev,?struct?
device_attribute* attr, char*?buf);
36.    static?ssize_t?virtualio_ reg_store(struct?device*?dev,?struct?
device_attribute*?attr,?const?char*?buf);
37.    ??
38.    /*定义设备属性*/??
39.
```

下面代码清单 14-3，列举了传统设备访问方法的具体实现。

代码清单 14-3　virtualio.c——传统设备访问实现部分

```
1.    /*打开设备方法*/
2.    static int virtualio_open(struct inode* inode, struct file*
filp) {
3.        struct virtualio_android_dev* dev;
4.
5.        /*将自定义设备结构体保存在文件指针的私有数据域中,以便访问设备时拿来用*/
6.        dev = container_of(inode->i_cdev, struct
virtualio_android_dev, dev);
7.        filp->private_data = dev;
8.
9.        return 0;
10.    }
11.
12.    /*设备文件释放时调用,空实现*/
13.    static int virtualio_release(struct inode* inode, struct file*
filp) {
14.        return 0;
15.    }
16.
17.    /*读取设备的寄存器val的值*/
18.    static ssize_t virtualio_read(struct file* filp, char __user
*buf,u8 num) {
19.        ssize_t err = 0;
20.        struct virtualio_android_dev* dev = filp->private_data;
```

```
21.
22.         /*同步访问*/
23.         if(down_interruptible(&(dev->sem))) {
24.             return -ERESTARTSYS;
25.         }
26.
27.         if(num <1 ||num>4) {
28.             goto out;
29.         }
30.
31.         /*将寄存器的值复制到用户提供的缓冲区*/
32.     switch (num)
33.       {
34.         case 1:
35.     copy_to_user(buf, &(dev->reg1), 1);
36.           break;
37.         case 2:
38.     copy_to_user(buf+1, &(dev->reg2), 1);
39.           break;
40.         case 3:
41.     copy_to_user(buf+2, &(dev->reg2), 1);
42.           break;
43.         case 4:
44.     copy_to_user(buf+3, &(dev->reg3), 1);
45.           break;
46.         default:
47.     err = -EFAULT;
48.             goto out;
49.           break;
50.       }
51.       err = num;
52.     out:
53.         up(&(dev->sem));
54.         return err;
55.     }
56.
57.     /*写设备的寄存器值*/
58.     static ssize_t virtualio_write(struct file* filp, const char
__user *buf, u8 num) {
59.         struct virtualio_android_dev* dev = filp->private_data;
60.         ssize_t err = 0;
61.
62.         /*同步访问*/
63.         if(down_interruptible(&(dev->sem))) {
64.             return -ERESTARTSYS;
65.         }
66.
67.         if(num <1 || num>4) {
68.             goto out;
69.         }
70.
71.         /*将用户提供的缓冲区的值写到设备寄存器去*/
```

```
72.       switch (num)
73.       {
74.         case 1:
75.     copy_from_user(&(dev->reg1), buf + num -1, 1);
76.           break;
77.         case 2:
78.     copy_from_user(&(dev->reg2), buf + num -1, 1);
79.           break;
80.         case 3:
81.     copy_from_user(&(dev->reg3), buf + num -1, 1);
82.           break;
83.         case 4:
84.     copy_from_user(&(dev->reg4), buf + num -1, 1);
85.           break;
86.         default:
87.     err = -EFAULT;
88.             goto out;
89.           break;
90.       }
91.       err = num;
92.
93.     out:
94.         up(&(dev->sem));
95.         return err;
96.     }
```

代码清单 14-4中定义了 devfs文件系统访问方法，这里把设备的寄存器reg1~4看成是设备的属性，通过读写这个属性来对设备进行访问。这里主要是通过实现 virtualio_reg_show和 virtualio_reg_store两个方法，以及定义了两个内部使用的访问 reg值的方法 __virtualio_get_reg和__virtualio_set_reg来实现。

代码清单 14-4　virtualio.c—— devfs文件系统访问方法实现部分

```
1.     /*读取寄存器的值到缓冲区buf中,内部使用*/
2.     static ssize_t __virtualio_get_reg(struct virtualio_android_dev*
dev, char* buf) {
3.         u8 reg1_val = 0;
4.     u8 reg2_val = 0;
5.     u8 reg3_val = 0;
6.     u8 reg4_val = 0;
7.
8.         /*同步访问*/
9.         if(down_interruptible(&(dev->sem))) {
10.             return -ERESTARTSYS;
11.         }
12.
```

```
13.         reg1_val= dev->reg1;
14.       reg2_val= dev->reg2;
15.       reg3_val= dev->reg3;
16.       reg4_val= dev->reg4;
17.         up(&(dev->sem));
18.
19.         return snprintf(buf, PAGE_SIZE, "%d, %d, %d, %d\n",
reg1_val, reg2_val, reg3_val,reg4_val);
20.     }
21.
22.     /*把缓冲区buf的值写到设备寄存器val中去,内部使用*/
23.     static ssize_t __virtualio_set_reg(struct
virtualio_android_dev* dev, const char buf[]) {
24.        u8 reg1_val = 0;
25.     u8 reg2_val = 0;
26.     u8 reg3_val = 0;
27.     u8 reg4_val = 0;
28.
29.         /*将字符串转换成数字*/
30.       reg1_val  = buf[0];
31.       reg2_val  = buf[1];
32.       reg3_val  = buf[2];
33.       reg4_val  = buf[3];
34.
35.         /*同步访问*/
36.         if(down_interruptible(&(dev->sem))) {
37.             return -ERESTARTSYS;
38.         }
39.
40.         dev->reg1 = reg1_val;
41.     dev->reg2 = reg2_val;
42.     dev->reg3 = reg3_val;
43.     dev->reg4 = reg4_val;
44.         up(&(dev->sem));
45.
46.         return count;
47.     }
48.
49.     /*读取设备属性*/
50.     static ssize_t virtualio_reg_show(struct device* dev, struct
device_attribute* attr, char* buf) {
51.         struct virtualio_android_dev* hdev = (struct
virtualio_android_dev*)dev_get_drvdata(dev);
52.
53.         return __virtualio_get_reg(hdev, buf);
54.     }
55.
56.     /*写设备属性*/
57.     static ssize_t virtualio_reg_store(struct device* dev, struct
device_attribute* attr, const char* buf, size_t count) {
58.         struct virtualio_android_dev* hdev = (struct
virtualio_android_dev*)dev_get_drvdata(dev);
```

```
59.
60.         return __virtualio_set_reg(hdev, buf);
61.     }
```

代码清单 14-5中定义了 proc文件系统访问方法，主要实现了 virtualio_proc_read和 virtualio_proc_write两个方法，同时定义了在proc文件系统创建和删除文件的方法 virtualio_create_proc和 virtualio_remove_proc。

代码清单 14-5　virtualio.c—— proc文件系统访问方法实现部分

```
1.     /*读取设备寄存器的值,保存在page缓冲区中*/
2.     static ssize_t virtualio_proc_read(char* page, char** start,
off_t off, int count, int* eof, void* data) {
3.         if(off > 0) {
4.             *eof = 1;
5.             return 0;
6.         }
7.
8.         return __virtualio_get_reg(hello_dev, page);
9.     }
10.
11.     /*把缓冲区的值buff保存到设备寄存器中*/
12.     static ssize_t virtualio_proc_write(struct file* filp, const
char __user *buff, unsigned long len, void* data) {
13.         int err = 0;
14.         char* page = NULL;
15.
16.         if(len > PAGE_SIZE) {
17.             printk(KERN_ALERT"The buff is too large: %lu.\n", len);
18.             return -EFAULT;
19.         }
20.
21.         page = (char*)__get_free_page(GFP_KERNEL);
22.         if(!page) {
23.             printk(KERN_ALERT"Failed to alloc page.\n");
24.             return -ENOMEM;
25.         }
26.
27.         /*先把用户提供的缓冲区值复制到内核缓冲区中*/
28.         if(copy_from_user(page, buff, len)) {
29.             printk(KERN_ALERT"Failed to copy buff from user.\n");
30.             err = -EFAULT;
31.             goto out;
32.         }
33.
34.         err = __virtualio_set_reg (virtualio_dev, page, len);
```

```
35.
36.    out:
37.        free_page((unsigned long)page);
38.        return err;
39.    }
40.
41.    /*创建/proc/virtualio文件*/
42.    static void virtualio_create_proc(void) {
43.        struct proc_dir_entry* entry;
44.
45.        entry = create_proc_entry(VIRTUALIO_DEVICE_PROC_NAME, 0,
NULL);
46.        if(entry) {
47.            entry->owner = THIS_MODULE;
48.            entry->read_proc = virtualio_proc_read;
49.            entry->write_proc = virtualio_proc_write;
50.        }
51.    }
52.
53.    /*删除/proc/virtualio文件*/
54.    static void virtualio_remove_proc(void) {
55.        remove_proc_entry(VIRTUALIO_DEVICE_PROC_NAME, NULL);
56.    }
```

为了让读者能更清楚地阅读，这里还给出定义模块加载和卸载方法，这里主要是执行设备注册和初始化操作，如代码清单 14-6。

代码清单 14-6　virtualio.c——设备驱动注册与注销部分

```
1.    /*初始化设备*/
2.    static int  __virtualio_setup_dev(struct virtualio_android_dev*
dev) {
3.        int err;
4.        dev_t devno = MKDEV(virtualio_major, virtualio_minor);
5.
6.        memset(dev, 0, sizeof(struct virtualio_android_dev));
7.
8.        cdev_init(&(dev->dev), &virtualio_fops);
9.        dev->dev.owner = THIS_MODULE;
10.        dev->dev.ops = &virtualio_fops;
11.
12.        /*注册字符设备*/
13.        err = cdev_add(&(dev->dev),devno, 1);
14.        if(err) {
15.            return err;
16.        }
17.
```

```
18.         /*初始化信号量和寄存器val的值*/
19.         init_MUTEX(&(dev->sem));
20.     dev->reg1 = 0;
21.     dev->reg2 = 0;
22.     dev->reg3 = 0;
23.     dev->reg4 = 0;
24.
25.         return 0;
26.     }
27.
28.     /*模块加载方法*/
29.     static int __init virtualio_init(void){
30.         int err = -1;
31.         dev_t dev = 0;
32.         struct device* temp = NULL;
33.
34.         printk(KERN_ALERT"Initializing virtualio device.\n");
35.
36.         /*动态分配主设备和从设备号*/
37.         err = alloc_chrdev_region(&dev, 0, 1,
VIRTUALIO_DEVICE_NODE_NAME);
38.         if(err < 0) {
39.             printk(KERN_ALERT"Failed to alloc char dev region.\n");
40.             goto fail;
41.         }
42.
43.         virtualio_major = MAJOR(dev);
44.         virtualio_minor = MINOR(dev);
45.
46.         /*分配helo设备结构体变量*/
47.         virtualio_dev = kmalloc(sizeof(struct hello_android_dev),
GFP_KERNEL);
48.         if(!virtualio_dev) {
49.             err = -ENOMEM;
50.             printk(KERN_ALERT"Failed to alloc virtualio_dev.\n");
51.             goto unregister;
52.         }
53.
54.         /*初始化设备*/
55.         err = __virtualio_setup_dev(virtualio_dev);
56.         if(err) {
57.             printk(KERN_ALERT"Failed to setup dev: %d.\n", err);
58.             goto cleanup;
59.         }
60.
61.         /*在/sys/class/目录下创建设备类别目录virtualio*/
62.         virtualio_class = class_create(THIS_MODULE,
VIRTUALIO_DEVICE_CLASS_NAME);
63.         if(IS_ERR(virtualio_class)) {
64.             err = PTR_ERR(virtualio_class);
65.             printk(KERN_ALERT"Failed to create virtualio
class.\n");
```

```
66.             goto destroy_cdev;
67.         }
68.
69.         /*在/dev/目录和/sys/class/virtualio目录下分别创建设备文件
virtualio*/
70.         temp = device_create(virtualio_class, NULL, dev, "%s",
VIRTUALIO_DEVICE_FILE_NAME);
71.         if(IS_ERR(temp)) {
72.             err = PTR_ERR(temp);
73.             printk(KERN_ALERT"Failed to create virtualio device.");
74.             goto destroy_class;
75.         }
76.
77.         dev_set_drvdata(temp, virtualio_dev);
78.
79.         /*创建/proc/virtualio文件*/
80.         virtualio_create_proc();
81.
82.         printk(KERN_ALERT"Succedded to initialize virtualio
device.\n");
83.         return 0;
84.
85.     destroy_device:
86.         device_destroy(virtualio_class, dev);
87.
88.     destroy_class:
89.         class_destroy(virtualio_class);
90.
91.     destroy_cdev:
92.         cdev_del(&(virtualio_dev->dev));
93.
94.     cleanup:
95.         kfree(virtualio_dev);
96.
97.     unregister:
98.         unregister_chrdev_region(MKDEV(virtualio_major,
virtualio_minor), 1);
99.
100.     fail:
101.         return err;
102.     }
103.
104.     /*模块卸载方法*/
105.     static void __exit virtualio_exit(void) {
106.         dev_t devno = MKDEV(virtualio_major, virtualio_minor);
107.
108.         printk(KERN_ALERT"Destroy virtualio device.\n");
109.
110.         /*删除/proc/virtualio文件*/
111.         virtualio_remove_proc();
112.
113.         /*销毁设备类别和设备*/
```

```
114.         if(virtualio_class) {
115.             device_destroy(virtualio_class, MKDEV(virtualio_major,
virtualio_minor));
116.             class_destroy(virtualio_class);
117.         }
118.
119.         /*删除字符设备和释放设备内存*/
120.         if(virtualio_dev) {
121.             cdev_del(&(virtualio_dev->dev));
122.             kfree(virtualio_dev);
123.         }
124.
125.         /*释放设备号*/
126.         unregister_chrdev_region(devno, 1);
127.     }
128.
129.     MODULE_LICENSE("GPL");
130.     MODULE_DESCRIPTION("First Android Driver");
131.
132.     module_init(virtualio_init);
133.     module_exit(virtualio_exit);
```

5）在代码编写完后，在 virtualio目录中新增 Kconfig和 Makefile两个文件。前面讲过， Kconfig是在编译前执行配置命令 make menuconfig时用到的，而 Makefile是执行编译命令 make时用到的。

代码清单 14-7　Kconfig文件的内容

```
config VIRTUALIO
    tristate "First Android Driver"
    default n
    help
    This is the first android driver.
```

代码清单 14-8　Makefile文件的内容

```
obj-$（CONFIG_VIRTUALIO）+=virtualio.o
```

在 Kconfig文件中， tristate表示编译选项 VIRTUALIO支持在编译内核时，virtualio模块支持模块、内建和不编译三种编译方法，默认是不编译，因

此，在编译内核前，我们还需要执行 make menuconfig命令来配置编译选项，使得 virtualio模块可以以模块或者内建的方法进行编译。

在 Makefile文件中，根据选项 VIRTUALIO的值，决定是否要编译 virtualio驱动模块。

6）修改 arch/arm/Kconfig和 drivers/kconfig两个文件，在 menu"Device Drivers"和 endmenu之间添加一行：

```
source"drivers/virtualio/Kconfig"
```

这样，执行 make menuconfig时就可以配置 virtualio模块的编译选项了。

7）修改 drivers/Makefile文件，添加一行：

```
obj-$（CONFIG_VIRTUALIO）+=virtualio/
```

8）配置编译选项：

```
$make menuconfig
```

找到 "Device Drivers"=>"First Android Drivers"选项，设置为 y。

9）编译：

```
$make
```

编译成功后，就可以在 virtualio目录下看到 virtualio.o文件了，这时候编译出来的 zImage已经包含了 virtualio驱动。

14.3　Android集成 C程序访问 virtualio

经过上一节的准备，我们已开发好 virtualio设备驱动。下面我们将循序渐进地讲解 Android如何通过驱动来访问与使用该虚拟设备。在本节中，我们先通过驱动程序员都熟悉的 C程序来访问。

前面我们也讲过， Android中应用程序一般是用 Java语言开发的。但是 C语言也可以开发 Android中的可执行程序。下面我们将来看看如何通过 C程序来访问上一节中创建的 3个文件节点：传统的设备文件 /dev/virtualio、 proc系统文件 /proc/virtualio和 devfs系统属性文件 /sys/class/virtualio/virtualio/val。

1）进入 Android源代码工程的 external目录，创建 virtualio目录 ：

```
$ cd external
$ mkdir virtualio
```

2）在 virtualio目录中新建 virtualio.c文件。参见代码清单 14-9。

代码清单 14-9　external/virtualio/virtualio.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <fcntl.h>
4. #define DEVICE_NAME "/dev/virtualio"
5. int main(int argc, char** argv)
6. {
7.     int fd = -1;
8.     int reg_val = 0;
9.     int I;
10.     fd = open(DEVICE_NAME, O_RDWR);
11.     if(fd == -1) {
12.         printf("Failed to open device %s.\n", DEVICE_NAME);
13.         return -1;
14.     }
15.
16.     printf("Read original value:\n");
```

```
17.      for(i = 0; i< 4 ; i++)
18.      {
19.          read(fd, &reg_val, i+1);
20.          printf("%d.\n", reg_val);
21.      }
22.      for(i=0; i<4; i++)
23.      {
24.          reg_val = 6+i;
25.          printf("Write value %d to %s.\n\n", reg_val,
DEVICE_NAME);
26.          write(fd, &reg_val, i+1);
27.      }
28.
29.          printf("Read the value again:\n");
30.      for(i = 0; i< 4 ; i++)
31.      {
32.          read(fd, &reg_val, i+1);
33.          printf("%d.\n", reg_val);
34.      }
35.      close(fd);
36.      return 0;
37. }
```

3）在 virtualio目录中新建 Android.mk文件。参见代码清单 14-10。

代码清单 14-10　external/virtualio/Android.mk

```
LOCAL_PATH := $(call my-dir)
     include $(CLEAR_VARS)
     LOCAL_MODULE_TAGS := optional
     LOCAL_MODULE := virtualio
     LOCAL_SRC_FILES := $(call all-subdir-c-files)
     include $(BUILD_EXECUTABLE)
```

其中，最后一句中的 BUILD_EXECUTABLE表示我们要编译的是可执行程序。

4）使用 mmm命令进行编译 virtualio C程序模块：

```
$mmm./external/virtualio
```

编译成功后，就可以在 out/target/product/gerneric/system/bin目录下，看到可执行文件 virtualio了。

5）重新打包 Android系统文件 system.img：

```
$make snod
```

这样，重新打包后的 system.img文件就包含刚才编译好的 virtualio可执行文件了。

6）运行烧写新 system.img的机器或模拟器，可执行 /system/bin/virtualio C程序来访问 14.2节中开发的内核驱动程序：

```
$ adb shell        // 通过PC链接上Android机器,以便访问该类机器
root@android:/ # cd system/bin        //这是登入Android机器后,对该类机器操作
root@android:/system/bin # ./virtualio
Read the original value:
0.
0.
0.
0.
Write value 5 to /dev/ virtualio.
Read the value again:
6.
7.
8.
9.
```

从执行结果来看，通过该 C程序，我们已可以顺利地访问 Android内核驱动了。

14.4　Android通过 HAL访问 virtualio

14.4.1　virtualio HAL模块实现

虽然 14.3节中，通过纯 C语言的方法可以成功地编写访问 Android底层驱动的程序，但这种方法太简单直白，没有很好地利用 Android架构，如 HAL带来的好处。从本节开始，我们将讲解如何在 Android的HAL中，增加与硬件相关的模块，进而实现与底层驱动的交互。另外，我们还将顺便讲解如何在 Android系统创建设备文件时，用类似一般 Linux udev规则修改文件模式的方法。

1）在 hardware/libhardware/include/hardware目录，新建 virtualio.h文件。代码清单参见 14-11。

```
$ cd?hardware/libhardware/include/hardware
$ vi virtualio.h
```

代码清单 14-11
hardware/libhardware/include/hardware/virtualio.h

```
1. #ifndef ANDROID_VIRTUALIO_INTERFACE_H
2. #define ANDROID_VIRTUALIO_INTERFACE_H
3. #include <hardware/hardware.h>
4.
5. __BEGIN_DECLS
6.
7. /*定义模块ID*/
8. #define VIRTUALIO_HARDWARE_MODULE_ID "virtualio"
9.
10. /*硬件模块结构体*/
11. struct virtualio_module_t {
12.     struct hw_module_t common;
13. };
14.
15. /*硬件接口结构体*/
16. struct virtualio_device_t {
17.     struct hw_device_t common;
```

```
18.     int fd;
19.     int (*set_reg)(struct virtualio_device_t* dev, char reg);
20.     int (*get_ reg)(struct virtualio_device_t* dev, char*
reg);
21. };
22.
23. __END_DECLS
24.
25. #endif
```

其中，根据 Android硬件抽象层规范的要求，分别定义模块 ID、硬件模块结构体以及硬件接口结构体。在硬件接口结构体中， fd表示设备文件描述符，对应我们将要处理的设备文件 ″/dev/virtualio″，set_reg和 get_reg为该 HAL对上提供的函数接口。

2）进入 hardware/libhardware/modules目录，新建 virtualio目录，并添加 virtualio.c文件。参见代码清单 14-12～ 14-14。

代码清单 14-12
hardware/libhardware/modules/virtualio/virtualio.c——结构定义部分

```
1. #define LOG_TAG "VirtualioStub"
2.
3. #include <hardware/hardware.h>
4. #include <hardware/virtualio.h>
5. #include <fcntl.h>
6. #include <errno.h>
7. #include <cutils/log.h>
8. #include <cutils/atomic.h>
9.
10. #define DEVICE_NAME "/dev/virtualio"
11. #define MODULE_NAME "Virtualio"
12. #define MODULE_AUTHOR "mobilephonesoft@gmail.com"
13.
14. /*设备打开和关闭接口*/
15. static int virtualio_device_open(const struct hw_module_t*
module, const char* name, struct hw_device_t** device);
16. static int virtualio_device_close(struct hw_device_t*
device);
17.
18. /*设备访问接口*/
```

```
19. static int virtualio_set_reg(struct virtualio_device_t* dev,
char reg, unsigned char num);
20. static int virtualio_get_reg(struct virtualio_device_t*
dev,char*reg, unsigned char num);
21.
22. /*模块方法表*/
23. static struct hw_module_methods_t virtualio_module_methods =
{
24.     open: virtualio_device_open
25. };
26.
27. /*模块实例变量*/
28. struct virtualio_module_t HAL_MODULE_INFO_SYM = {
29.     common: {
30.         tag: HARDWARE_MODULE_TAG,
31.         version_major: 1,
32.         version_minor: 0,
33.         id: VIRTUALIO_HARDWARE_MODULE_ID,
34.         name: MODULE_NAME,
35.         author: MODULE_AUTHOR,
36.         methods: &virtualio_module_methods,
37.     }
38. };
```

其中，实例变量名必须为 HAL_MODULE_INFO_SYM， tag也必须为 HARDWARE_MODULE_TAG，这是 Android硬件抽象层规范规定的。

代码清单 14-13 hardware/libhardware/modules/virtualio/virtualio.c—— open函数实现部分

```
1. static int virtualio_device_open(const struct hw_module_t*
module, const char* name, struct hw_device_t** device) {
2.     struct virtualio_device_t* dev;
3.
4. dev = (struct virtualio_device_t*)malloc(sizeof(struct
virtualio_device_t));
5.     if(!dev) {
6.         LOGE("Virtualio Stub: failed to alloc space");
7.         return -EFAULT;
8.     }
9.
10.     memset(dev, 0, sizeof(struct hello_device_t));
11.     dev->common.tag = HARDWARE_DEVICE_TAG;
```

```
12.     dev->common.version = 0;
13.     dev->common.module = (hw_module_t*)module;
14.     dev->common.close = virtualio_device_close;
15.     dev->set_reg = virtualio_set_reg;
16. dev->get_reg = virtualio_get_reg;
17.
18.     if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
19.         LOGE("Virtualio Stub: failed to open /dev/virtualio --
- %s.", strerror(errno));free(dev);
20.         return -EFAULT;
21.     }
22.
23.     *device = &(dev->common);
24.     LOGI("Virtualio Stub: open /dev/virtualio
successfully.");
25.
26.     return 0;
27. }
```

注意，由于设备文件是在内核驱动里面通过 device_create创建的，而device_create创建的设备文件默认只有 root用户可读写，而virtual_device_open一般是由上层应用程序来调用的，这些应用程序一般不具有 root权限，这时候应该会导致打开设备文件失败：

```
Virtualio Stub:failed to open/dev/virtualio--Permission denied.
```

解决办法是基于类似于 Linux的 udev规则，在 Android源代码工程目录，进入 system/core/rootdir目录，打开里面一个名为 ueventd.rc文件，往里面添加一行，以赋予其他用户对该文件的读 /写权限：

```
/dev/virtualio 0666 root root
```

代码清单 14-14 hardware/libhardware/modules/virtualio/virtualio.c—— set等函数实现部分

```
1. static int virtualio_device_close(struct hw_device_t* device)
{
2.     struct virtualio_device_t* virtualio_device = (struct
virtualio_device_t*)device;
3.
4.     if(virtualio_device) {
5.         close(virtualio_device->fd);
6.         free(virtualio_device);
7.     }
8.
9.     return 0;
10. }
11.
12. static int virtualio_set_reg(struct virtualio_device_t* dev,
char reg, unsigned char num) {
13.     LOGI("Virtualio Stub: set value %d to device reg.", reg);
14.
15.     write(dev->fd, &reg, num);
16.
17.     return 0;
18. }
19.
20. static int virtualio_get_reg(struct virtualio_device_t* dev,
char* reg, unsigned char num) {
21.     if(!reg) {
22.         LOGE("Virtualio Stub: error reg pointer");
23.         return -EFAULT;
24.     }
25.
26.     read(dev->fd, val, num);
27.
28.     LOGI("Virtualio Stub: get value %d from devicereg",
*reg);
29.
30.     return 0;
31. }
```

3）继续在 virtualio目录下新建 Android.mk文件，参见代码清单 14-15。

代码清单 14-15
hardware/libhardware/modules/virtualio/Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := virtualio.c
LOCAL_MODULE := virtualio.default
include $(BUILD_SHARED_LIBRARY)
```

注意， LOCAL_MODULE的定义规则， virtualio后面跟有 default。这样，就能够保证我们的模块 virtualio默认要被硬件抽象层加载。

4）编译上面编写完的 HAL模块：

```
$mmm hardware/libhardware/modules/virtualio
```

编译成功后，就可以在 out/target/product/generic/system/lib/hw目录下看到 virtualio.default.so文件了。

5）重新打包 Android系统镜像 system.img。

重新打包后， system.img就包含我们定义的硬件抽象层模块 virtualio.default了。

虽然我们在 Android系统为我们自己的硬件增加了一个硬件抽象层模块，但是现在 Java应用程序还不能访问到该硬件。我们还必须编写JNI方法和在 Android的 Application Framework层增加 API接口，才能让上层应用程序访问我们的硬件。接下来，我们还将完成这一系统过程，使得我们能够在 Java应用程序中访问我们自己定制的硬件。

14.4.2 实现访问 virtualio HAL模块 JNI

在 14.4.1节实现了 virtualio的 HAL层。实现该层就是要向 Android的 Application Framework层提供 virtualio的硬件服务。同时，前面我们也讲过 Android系统的应用程序是用 Java语言编写的，而硬件驱动程序是用 C语言来实现的，为此我们要实现 JNI方法调用，以便

Android中的 Java应用程序通过 JNI来调用硬件抽象层接口。在本节中，我们将讲解如何为 Android硬件抽象层接口编写 JNI方法，以便使得上层的 Java应用程序能够使用下层提供的硬件服务。

1）进入 frameworks/base/services/jni目录，新建 com_android_server_VirtualioService.cpp文件。

```
$ cd?frameworks/base/services/jni
$ vi?com_android_server_VirtualioService.cpp
```

在 com_android_server_VirtualioService.cpp文件中，实现 JNI方法。注意文件的命令方法， com_android_server前缀表示的是包名，表示硬件服务 VirtualioService是放在 frameworks/base/services/java目录下的 com/android/server目录的，即存在一个命名为 com.android.server.VirtualioService的类。关于 VirtualioService类，在下一小节中我们将做更详细的讲解。这里，我们只要知道 VirtualioService是一个提供 Java接口的硬件访问服务类。

com_android_server_VirtualioService.cpp代码清单参见清单 14-16。

代码清单 14-16　com_android_server_VirtualioService.cpp

```
1. #define LOG_TAG "VirtualioService"
2. #include "jni.h"
3. #include "JNIHelp.h"
4. #include "android_runtime/AndroidRuntime.h"
5. #include <utils/misc.h>
6. #include <utils/Log.h>
7. #include <hardware/hardware.h>
8. #include <hardware/virtualio.h>
9. #include <stdio.h>
10. namespace android
11. {
12.     /*在硬件抽象层中定义的硬件访问结构体,参考
<hardware/virtualio.h>*/
13.         struct virtualio_device_t* virtualio_device = NULL;
```

```
14.      /*通过硬件抽象层定义的硬件访问接口设置硬件寄存器的值*/
15.          static void virtualio_setReg(JNIEnv* env, jobject
clazz, jchar value, jcharnum) {
16.          char reg = value;
17.          LOGI("Virtualio JNI: set value %d to device reg.",
reg);
18.          if(!virtualio_device) {
19.              LOGI("Virtualio JNI: device is not open.");
20.              return;
21.          }
22.
23.          virtualio_device->set_reg(virtualio_device, reg,
num);
24.      }
25.          /*通过硬件抽象层定义的硬件访问接口读取硬件寄存器的值*/
26.      static jint virtualio_getReg(JNIEnv* env, jobject clazz,
jcharnum) {
27.          char reg= 0;
28.          if(!virtualio_device) {
29.              LOGI("Virtualio JNI: device is not open.");
30.              return reg;
31.          }
32.          virtualio_device->get_reg(virtualio_device, &reg,
num);
33.
34.          LOGI("Virtualio JNI: get value %d from device reg.",
reg);
35.
36.          return reg;
37.      }
38.          /*通过硬件抽象层定义的硬件模块打开接口打开硬件设备*/
39.      static inline int virtualio_device_open(const
hw_module_t* module, struct virtualio_device_t** device) {
40.          return module->methods->open(module,
VIRTUALIO_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
41.      }
42.          /*通过硬件模块ID来加载指定的硬件抽象层模块并打开硬件*/
43.      static jboolean virtualio_init(JNIEnv* env, jclass clazz)
{
44.          virtualio_module_t* module;
45.
46.          LOGI("Virtualio JNI: initializing...");
47.          if(hw_get_module(VIRTUALIO_HARDWARE_MODULE_ID, (const
struct hw_module_t**)&module) == 0) {
48.              LOGI("Virtualio JNI: virtualio Stub found.");
49.              if(virtualio_device_open(&(module->common),
&virtualio_device) == 0) {
50.                  LOGI("Virtualio JNI: virtualio device is
```

```
open.");
51.                 return 0;
52.             }
53.             LOGE("Virtualio JNI: failed to open virtualio
device.");
54.             return -1;
55.         }
56.         LOGE("Virtualio JNI: failed to get virtualio stub
module.");
57.         return -1;
58.     }
59.         /*JNI方法表*/
60.     static const JNINativeMethod method_table[] = {
61.         {"init_native", "()Z", (void*)virtualio_init},
62.         {"setReg_native", "(I)V", (void*)virtualio_setReg},
63.         {"getReg_native", "()I", (void*)virtualio_getReg},
64.     };
65.         /*注册JNI方法*/
66.     int register_android_server_VirtualioService(JNIEnv *env)
{
67.             return jniRegisterNativeMethods(env,
"com/android/server/VirtualioService", method_table,
NELEM(method_table));
68.     }
69. };
```

注意，在 virtualio_init函数中，通过 Android硬件抽象层提供的 hw_get_module方法来加载模块 ID为 VIRUTALIO_HARDWARE_MODULE_ID 的硬件抽象层模块，其中， VIRTUALIO_HARDWARE_MODULE_ID是在 hardware/virtualio.h中定义的。 Android硬件抽象层会根据 VIRTUALIO_HARDWARE_MODULE_ID的值在 Android系统的 /system/lib/hw目录中找到相应的模块，然后加载并且返回 hw_module_t接口给调用者使用。在 jniRegisterNativeMethods函数中，第二个参数的值必须对应 VirtualioService所在的包的路径，即 com.android.server.VirtualioService。

2）修改 frameworks/base/services/jni目录下的 onload.cpp文件，参见代码清单 14-17— 14-18。

代码清单 14-17 修改 namespace android

```
namespace android {
```

```
    ...
    int register_android_server_VirtualioService(JNIEnv *env);
// 增加注册服务接口声明
};
```

代码清单 14-18　修改 JNI_onLoad

```
extern "C" jint JNI_onLoad(JavaVM* vm, void* reserved)
{
    ...
    register_android_server_VirtualioService(env);        // 调用
注册服务接口函数
    ...
}
```

这样，在 Android系统初始化时就会自动加载该 JNI方法调用表。

3）修改同目录下的 Android.mk文件，在 LOCAL_SRC_FILES变量中增加一行。

```
LOCAL_SRC_FILES:= \
    com_android_server_AlarmManagerService.cpp \
    com_android_server_BatteryService.cpp \
    com_android_server_InputManager.cpp \
    com_android_server_LightsService.cpp \
    com_android_server_PowerManagerService.cpp \
    com_android_server_SystemServer.cpp \
    com_android_server_UsbService.cpp \
    com_android_server_VibratorService.cpp \
    com_android_server_location_GpsLocationProvider.cpp \
    com_android_server_VirtualioService.cpp \
    onload.cpp
```

4）编译并重新生成 system.img。

```
$ mmm?frameworks/base/services/jni
$ make snod
```

这样，重新打包的 system.img镜像文件就包含我们刚才编写的 JNI方法了，也就是我们可以通过 Android系统的 Application Framework层提供的硬件服务 VirtualioService来调用这些 JNI方法，进而调用低层的硬件抽象层接口去访问硬件了。

14.4.3 在 Framework层增加 virtualio服务

在实现 JNI接口方法后，为了让 Android Java应用程序真正使用到 virtualio的硬件服务，接下来我们就要完成最后一步：在 Android的 framework层实现相应的 VirtualioService服务。

1）在 Android系统中，硬件服务一般是运行在与应用程序相独立的进程中。因此，调用这些硬件服务的应用程序与这些硬件服务之间的通信需要通过代理来进行。为此，我们要首先定义好通信接口。进入 frameworks/base/core/java/android/os目录，新增 IVirtualioService.aidl接口定义文件，参见代码清单 14-19。

代码清单 14-19 IVirtualioService.aidl

```
1. package android.os;
2.
3. interface IVirtualioService {
4.     void setReg(int val);
5.     int getReg();
6. }
```

IVirtualioService接口提供了读和写 Virtualio设备硬件寄存器 reg1~4值的功能，分别通过 setReg和 getReg两个函数来实现。

2）返回到 frameworks/base目录，打开 Android.mk文件，修改 LOCAL_SRC_FILES变量的值，增加 IVirtualioService.aidl源文件。

```
LOCAL_SRC_FILES += \
     ...
     core/java/android/os/IVibratorService.aidl \
     core/java/android/os/IVirtualioService.aidl \
```

```
core/java/android/service/urlrenderer/IUrlRendererService.aidl \
     ...
```

3）编译 IVirtualioService.aidl接口。

```
$mmmframeworks/base
```

这样，就会根据 IVirtualioService.aidl生成相应的 IVirtualioService.Stub接口。

4）进入 frameworks/base/services/java/com/android/server目录，新增 IVirtualioService.java文件，参见代码清单 14-20。

代码清单 14-20　IVirtualioService.java

```
package com.android.server;
import android.content.Context;
import android.os.IVirtualioService;
import android.util.Slog;
public class VirtualioService extends IVirtualioService.Stub {
    private static final String TAG = "VirtualioService";
    VirtualioService() {
        init_native();
    }
    public void setReg(int val) {
        setReg_native(val);
    }
    public int getReg () {
        return getReg_native();
    }

    private static native boolean init_native();
        private static native void setReg_native(int val);
    private static native int getReg_native();
};
```

在 VirtualioService中，主要是通过调用 JNI方法 init_native、setReg_native和 getReg_native来提供 virtualio的硬件服务。

5）修改同目录的 SystemServer.java文件，在 ServerThread::run函数中增加加载 VirtualioService的代码。参见代码清单 14-21。

代码清单 14-21　修改 SystemServer.java

```
@Override
   public void run() {
   ...
       try {
          Slog.i(TAG, "DiskStats Service");
          ServiceManager.addService(?diskstats", new
DiskStatsService(context));
       } catch (Throwable e) {
          Slog.e(TAG, ?Failure starting DiskStats Service", e);
       }
       try {
          Slog.i(TAG, "Virtualio Service");
          ServiceManager.addService("virtualio", new
VirtualioService());
       } catch (Throwable e) {
          Slog.e(TAG, "Failure starting Virtualio Service", e);
       }
   ...
   }
```

6）编译 VirtualioService和重新打包 system.img

这样，重新打包后的 system.img系统镜像文件就在 Application Framework层中包含了我们自定义的硬件服务 VirtualioService了，并且会在系统启动的时候，自动加载 VirtualioService。到这里，应用程序就可以通过 Java接口来访问 Virtualio硬件服务了。

关于 Android Java应用程序的开发，这里就不讲了。有兴趣的读者可以查阅相应的书籍，为该硬件服务编写一个测试程序。另外，前面提到该硬件服务被实现为独立的进程，我们 Java应用程序要与之交互，就要借用 IPC技术；在 Android中实现了专有的 Binder驱动来实现这种进程间的通信。第 18章将更详细地讲解这个特殊的 Android驱动。

第15章　Framebuffer子系统

经过第14章的学习，我们了解了Android中的设备驱动工作机制。下面我们就要进入Android具体实用的设备驱动学习了。当然，我们也不可能涉及Android中所有设备驱动，所以还是请大家体会Android驱动的架构体系和设计思想，在实践新驱动开发并掌握相关知识基础上，能够举一反三，活学活用。

事实上，在Linux系统中，把不同类别的设备实现成了不同的子系统，而实际的设备驱动就是该子系统中的实例。Android系统中沿用了这个设计架构。同时，为了结合上层应用程序的使用，如果只用底层模块的概念来讲解一个系统运用驱动，很难完整地讲清楚，因此从这里开始我们引用了子系统的概念，本章要学习的Framebuffer子系统，就是为了显示输出而设计开发的。下面我们就来学习Android系统中这个最重要的输出是如何实现的。

15.1　Linux Framebuffer一般子系统

Framebuffer驱动在 Linux中用于系统最重要的输出——显示。以该驱动为基础， Linux机器得以向用户展现一个色彩斑斓的世界。对 PC而言，它就是显卡驱动；对嵌入式系统而言，它就是显示控制器和 LCD模组的驱动。

Framebuffer设备是一个字符类设备，它在文件系统中的设备节点通常是 /dev/fbX。当一个系统中有多个显示设备时，依次用 /dev/fb0、/dev/fb1等来表示。在 Android系统中，该类设备的主设备号为 29。Framebuffer设备中文又称作“帧缓冲设备”。

图 15-1是基于帧缓冲的 Framebuffer显示框架。

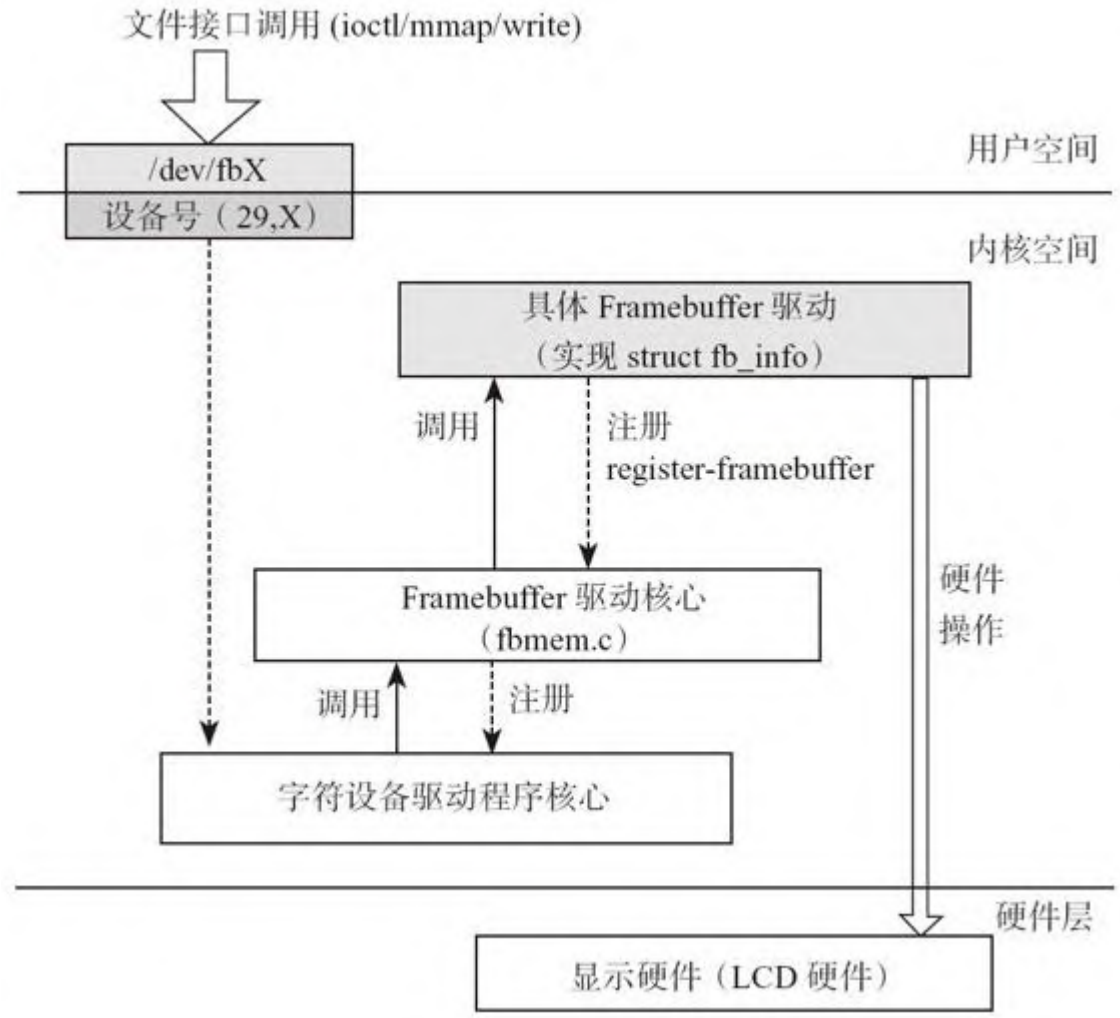


图15-1　Framebuffer显示驱动

15.1.1　Framebuffer数据结构

1.fb_info结构体

从图 15-1中可以看到，这个最重要的结构体其实就代表着framebuffer驱动，该结构体的定义参见代码清单 15-1。

代码清单 15-1fb_info结构体

```
struct fb_info
{
    int node;
    int flags;
    struct fb_var_screeninfo var; //可变参数
    struct fb_fix_screeninfo fix; //固定参数
    struct fb_monspecs monspecs; //显示器标准
    struct work_struct queue; // 帧缓冲事件队列
    struct fb_pixmap pixmap; //图像硬件mapper
    struct fb_pixmap sprite; // 光标硬件mapper
    struct fb_cmap cmap; //目前的颜色表
    struct list_head modelist;
    struct fb_videomode *mode; //目前的video模式

#ifdef CONFIG_FB_BACKLIGHT
    struct mutex bl_mutex;
    struct backlight_device *bl_dev; //对应的背光设备
    u8 bl_curve[FB_BACKLIGHT_LEVELS]; //背光调整
    #endif

struct fb_ops *fbops; //fb_ops,帧缓冲操作
    struct device *device;
    struct class_device *class_device
    int class_flag; //私有sysfs标志
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops; //图块Blitting
#endif
    char _ _iomem *screen_base; //虚拟基地址
    unsigned long screen_size;// ioremapped的虚拟内存大小
    void *pseudo_palette; //伪16色颜色表
```

```
#define FBINFO_STATE_RUNNING 0
#define FBINFO_STATE_SUSPENDED 1
    u32 state; //硬件状态,如挂起
    void *fbcon_par;
    void *par;
};
```

该结构体记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及操作函数指针。每一个帧缓冲设备都必须对应一个 fb_info结构实体。

2.fb_ops结构体

该结构体是 fb_info结构体的一个成员变量，指向显示操作的函数的指针，这些函数是需要驱动程序开发人员编写的，其定义如代码清单15-2所示。

代码清单 15-2　fb_ops结构体

```
struct fb_ops
{
struct module *owner;
/* 打开、释放*/
int(*fb_open)(struct fb_info *info, int user);
int(*fb_release)(struct fb_info *info, int user);

/* 对于非线性布局的或常规内存映射无法工作的帧缓冲设备需要*/
ssize_t(*fb_read)(struct file *file, char _ _user *buf, size_t
count, loff_t*ppos);
ssize_t(*fb_write)(struct file *file, const char _ _user
*buf,size_t count, loff_t *ppos);

/* 检测可变参数,并调整到支持的值*/
int(*fb_check_var)(struct fb_var_screeninfo *var, struct
fb_info *info);

/* 根据info->var设置video模式*/
int(*fb_set_par)(struct fb_info *info);

/* 设置color寄存器*/
int(*fb_setcolreg)(unsigned regno, unsigned red, unsigned
```

```
green,unsigned blue, unsigned transp, struct fb_info *info);

/* 批量设置color寄存器,设置颜色表*/
int(*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

/*显示空白*/
int(*fb_blank)(int blank, struct fb_info *info);

/* pan显示*/
int(*fb_pan_display)(struct fb_var_screeninfo *var, struct
fb_info *info);

/* 矩形填充*/
void(*fb_fillrect)(struct fb_info *info, const struct
fb_fillrect *rect);
/* 数据复制*/
void(*fb_copyarea)(struct fb_info *info, const struct
fb_copyarea *region);
/* 图形填充*/
void(*fb_imageblit)(struct fb_info *info, const struct fb_image
*image);

/* 绘制光标*/
int(*fb_cursor)(struct fb_info *info, struct fb_cursor
*cursor);

/* 旋转显示*/
void(*fb_rotate)(struct fb_info *info, int angle);

/* 等待blit空闲 (可选) */
int(*fb_sync)(struct fb_info *info);

/* fb特定的ioctl (可选) */
int(*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned
long arg);

/* 处理32位的compat ioctl (可选) */
int(*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
unsigned long arg);

/* fb特定的mmap */
int(*fb_mmap)(struct fb_info *info, struct vm_area_struct
*vma);

/* 保存目前的硬件状态*/
void(*fb_save_state)(struct fb_info *info);
```

```
/* 恢复被保存的硬件状态*/
void(*fb_restore_state)(struct fb_info *info);
};
```

该结构体中 fb_check_var（）成员函数用于检查可以修改的屏幕参数并调整到合适的值，而 fb_set_par（）则使得用户设置的屏幕参数在硬件上有效。

3.fb_var_screeninfo和 fb_fix_screeninfo结构体

这两个结构体也是 fb_info结构的成员变量。 fb_var_screeninfo记录用户可修改的显示控制器参数，包括屏幕分辨率和每个像素点的比特数。 fb_var_screeninfo中的 xres定义屏幕一行有多少个点， yres定义屏幕一列有多少个点， bits_per_pixel定义每个点用多少个字节表示。而 fb_fix_screeninfo中记录用户不能修改的显示控制器的参数，如屏幕缓冲区的物理地址、长度。当对帧缓冲设备进行映射操作的时候，就是从 fb_fix_screeninfo中取得缓冲区物理地址。上述数据成员都需要在驱动程序中初始化和设置。

fb_var_screeninfo和 fb_fix_screeninfo结构体的定义分别如代码清单 15-3和代码清单 15-4所示。

代码清单 15-3　fb_var_screeninfo结构体

```
struct fb_var_screeninfo
{
/* 可见解析度*/
_u32 xres;
_u32 yres;
/* 虚拟解析度*/
_u32 xres_virtual;
_u32 yres_virtual;
/* 虚拟到可见之间的偏移*/
_u32 xoffset;
_u32 yoffset;

_u32 bits_per_pixel; // 每像素位数,BPP
_u32 grayscale;//非0时指灰度
```

```
/* fb缓存的R\G\B位域*/
struct fb_bitfield red;
struct fb_bitfield green;
struct fb_bitfield blue;
struct fb_bitfield transp;//透明度

_u32 nonstd; // != 0 非标准像素格式

_ _u32 activate;

_ _u32 height;//高度
_ _u32 width; //宽度

_ _u32 accel_flags; // 看fb_info.flags

/* 定时：除了pixclock本身外,其他的都以像素时钟为单位*/
_ _u32 pixclock; // 像素时钟（皮秒）
_ _u32 left_margin; // 行左空白：从同步到绘图之间的延迟
_ _u32 right_margin; // 行右空白：从绘图到同步之间的延迟
_ _u32 upper_margin; //帧上空白：从同步到绘图之间的延迟
_ _u32 lower_margin; // 帧下空白：从绘图到同步之间的延迟
_ _u32 hsync_len; // 水平同步的长度
_ _u32 vsync_len; // 垂直同步的长度
_ _u32 sync;
_ _u32 vmode;
_ _u32 rotate; //顺时针旋转的角度
_ _u32 reserved[5]; //保留
};
```

代码清单 15-4　fb_fix_screeninfo结构体

```
struct fb_fix_screeninfo
{
char id[16]; //字符串形式的标识符
unsigned long smem_start; //fb缓存的开始位置
_ _u32 smem_len; //fb缓存的长度
_ _u32 type; // FB_TYPE_
_ _u32 type_aux; // 分界
_ _u32 visual; //显示的色彩模式
_ _u16 xpanstep; //如果没有硬件panning ,赋0
```

```
_ _u16 ypanstep;
_ _u16 ywrapstep;/
_ _u32 line_length; // 1行的字节数
unsigned long mmio_start; // 内存映射I/O的开始位置
_ _u32 mmio_len; //内存映射I/O的长度
_ _u32 accel;
_ _u16 reserved[3]; // 保留
};
```

4.fb_bitfield结构体

fb_bitfield结构体描述每一像素显示缓冲区的组织方式，包含位域偏移、位域长度和 MSB指示，如代码清单 15-5所示。

代码清单 15-5　fb_bitfield结构体

```
struct fb_bitfield
{
_ _u32 offset; // 位域偏移
_ _u32 length; // 位域长度
_ _u32 msb_right; // MSB
};
```

5.文件操作结构体

作为一种字符设备，帧缓冲设备的文件操作结构体定义于/linux/drivers/vedio/fbmem.c文件中，如代码清单 15-6所示。

代码清单 15-6　帧缓冲设备文件操作结构体

```
static struct file_operations fb_fops =
{
.owner = THIS_MODULE,
.read = fb_read, //读函数
.write = fb_write, //写函数
.ioctl = fb_ioctl, //I/O控制函数
#ifdef CONFIG_COMPAT
```

```
.compat_ioctl = fb_compat_ioctl,
#endif
.mmap = fb_mmap, //内存映射函数
.open = fb_open, //打开函数
.release = fb_release, //释放函数
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
.get_unmapped_area = get_fb_unmapped_area,
#endif
};
```

帧缓冲设备驱动的文件操作接口函数已经在 fbmem.c中被统一实现，一般不需要由驱动开发工程师再编写。关于 Framebuffer工程师要花精力的地方，将在下面讲述。

15.1.2 Framebuffer驱动

Framebuffer驱动的程序结构如图 15-2所示。

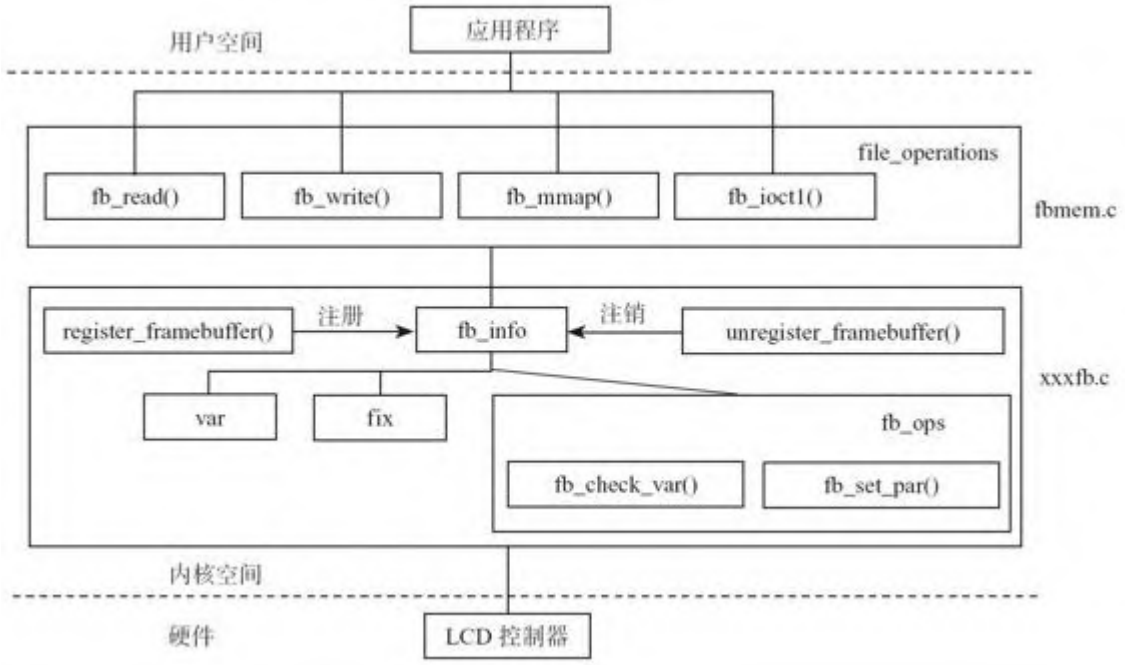


图15-2 Framebuffer驱动的程序结构

如图 15-2所示， Framebuffer的设备文件操作函数在 fbmem.c中实现。而特定于帧缓冲设备 fb_info结构体的注册、注销及 fb_info成员的维护，尤其是 fb_ops成员函数的实现，则在特定的 xxxfb.c源码

文件中落实。其中， fb_ops中的成员函数最终会操作显示控制器，以及 LCD模组上的寄存器，进而实现对显示设备的设置，以及向其输出显示数据。

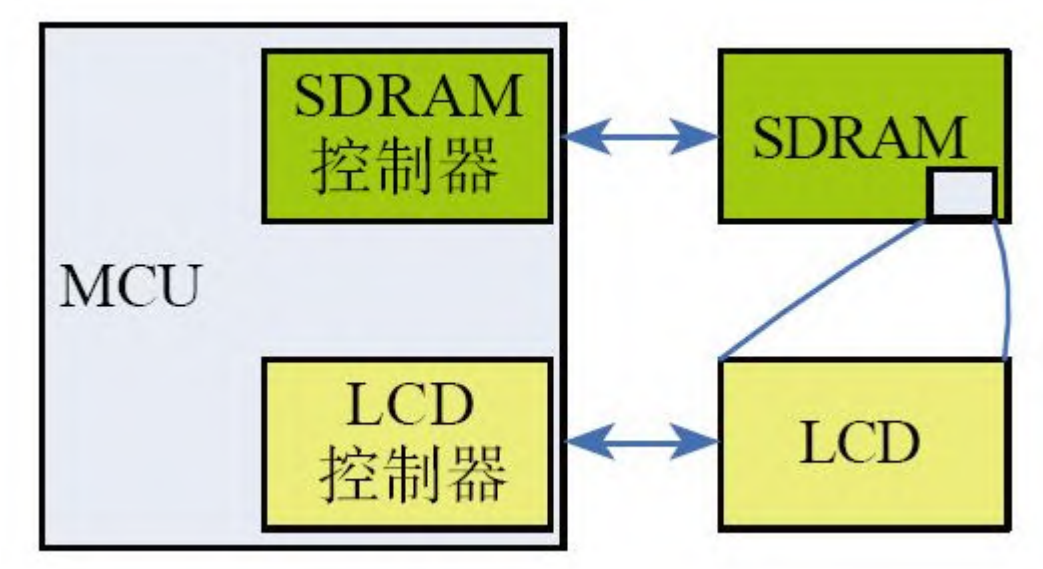


图 15-3　在 RAM中分配显存

在嵌入式系统中，常在 RAM中分配一段显示缓冲区，也就常说的显存，以存放要输出到显示设备上的显示数据，如图 15-3所示。

显示数据一般会采用 DMA方式来传输，因此分配显存一般由 dma_alloc_writecombine（）函数完成。该函数会在内存中分配一段 writecombining区域。而该内存的释放则由 dma_free_writecombine（）函数完成。显存的分配与释放，参见代码清单 15-7所示。

代码清单 15-7　帧缓冲设备显存的分配与释放

```
static int __init xxxfb_map_video_memory(struct xxxfb_info
*fbi)
{
fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
fbi->map_cpu = dma_alloc_writecombine(fbi->dev, fbi->map_size,
&fbi->map_dma,GFP_KERNEL); //分配内存
```

```
fbi->map_size = fbi->fb->fix.smem_len; //显示缓冲区大小

if (fbi->map_cpu)
{
memset(fbi->map_cpu, 0xf0, fbi->map_size);

fbi->screen_dma = fbi->map_dma;
fbi->fb->screen_base = fbi->map_cpu;
fbi->fb->fix.smem_start = fbi->screen_dma; //赋值fix 的
smem_start
}

return fbi->map_cpu ? 0 : - ENOMEM;
}

static inline void xxxfb_unmap_video_memory(struct
s3c2410fb_info *fbi)
{
dma_free_writecombine(fbi->dev,fbi->map_size,fbi->map_cpu, fbi-
>map_dma); //释放显示缓冲区
}
```

---

writecombining意味着“写合并”，它允许写入的数据被合并，并临时保存在被分配的写合并缓冲区中，直到进行一次突发传输（ burst传输）而不再需要多次单个传输（ single传输）。通过dma_alloc_writecombine（）分配的显示缓冲区不会出现 Cache一致性问题，这一点类似于 dma_alloc_coherent（）。

15.2　Android Framebuffer子系统实践

15.1节讲述了 Linux OS Framebuffer子系统普适性的内容，下面我们通过 Android中的具体帧缓冲设备来看看 Android基于 Framebuffer的应用与实践。

以下示例基于 Marvell PXA910CPU， T35KPS05E高清 LCD模组。

15.2.1　硬件基础

在这个例子中，具有 WVGA分辨率（ 480×800）的 LCD模组，通过 PXA910LCD控制器得到了 RGB888格式 24位真彩的图像数据（ LCD模组实际只取了 RGB666格式 18位的图像数据）；而且 PXA910通过 SPI总线，实现对 LCD模组工作模式选择、配置寄存器读写、调整 gamma值等。其硬件框架如图 15-4所示。

图 15-5是 PXA910LCD控制器的功能框图。该图左侧是与 CPU相连的 ARM高速总线（ AHB）以及高速扩展总线接口（ AXT），通过这两条总线， LCD控制器将分别接收来自 CPU的控制及相应的图像数据。该控制器将可以同时处理两路图像数据，通过对图像数据的混合、透明比例设置、色域变换、对比度、饱和度调整等，实现对图像数据的各种操作。可以说该控制器拥有了一般 PC显卡所需要的所有功能。处理后的图像数据将通过图 15-5右边的智能面板，向 LCD模组等图像输出设备，输出丰富的图像数据。

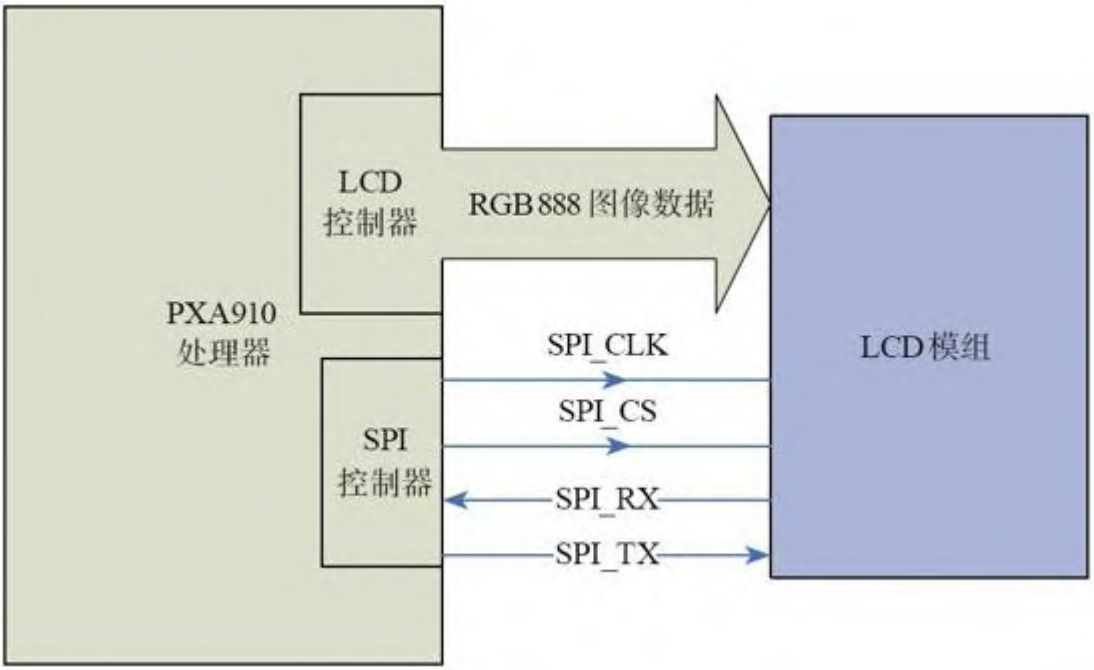


图15-4　PXA910与LCD模组接口框图

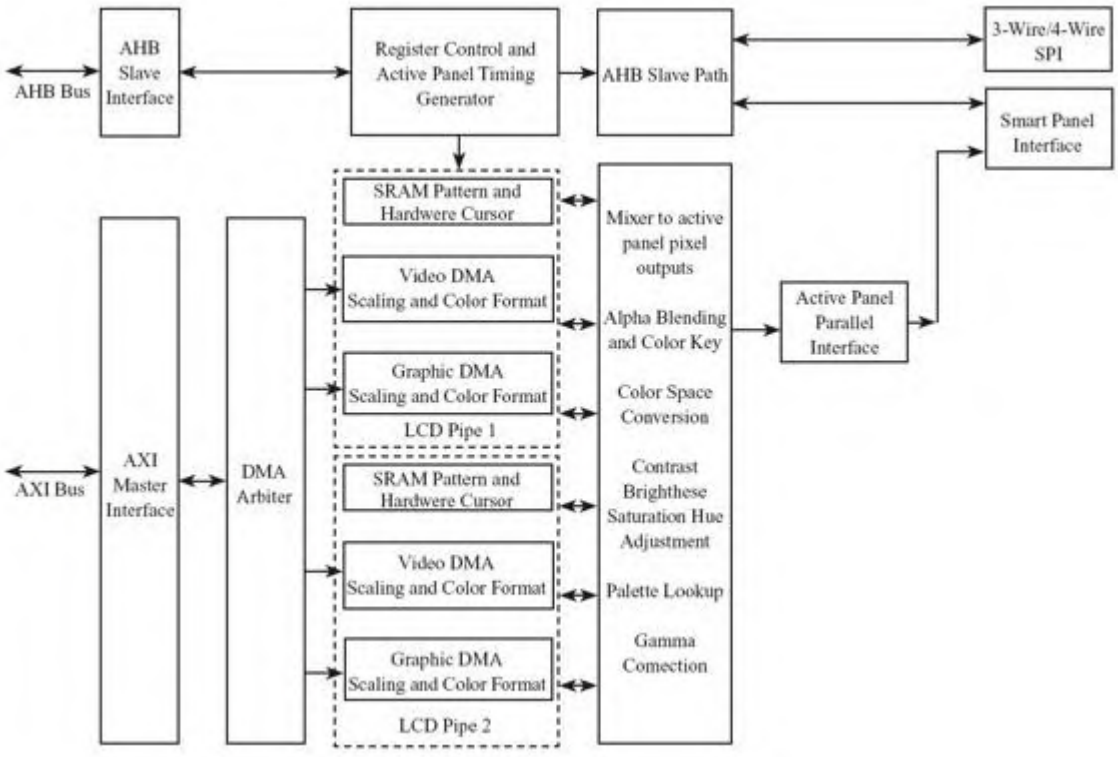


图15-5　PXA910LCD功能框图

其实在每路图像数据中又支持三个图层：第一层是图像层，用于显示一般静态的图形界面；第二层是视频层，用于显示一般的动态视频；第三层是光标层，用于显示光标等标识当前位置的图像元素。通过这三个图层的合理叠加，将满足智能终端丰富图像输出的需求，如图15-6所示。

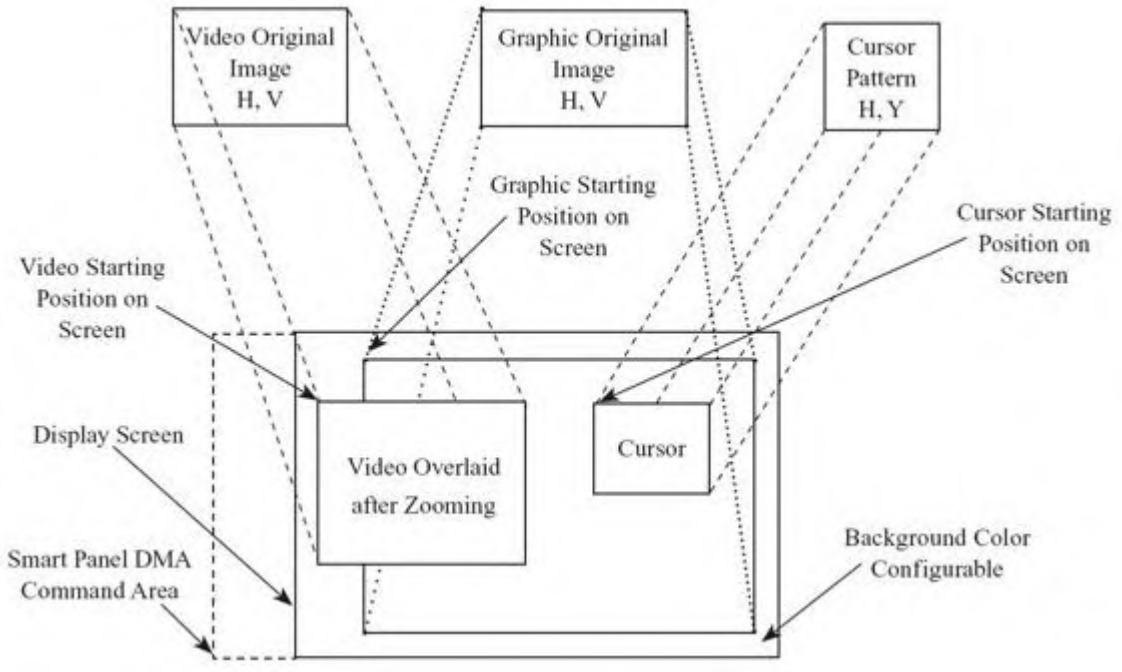


图15-6　PXA910一路显示数据的三个图层

T35KPS05E高清 LCD模组，除了具有 TFT显示驱动矩阵，还拥有480×800大小的 GRAM，以存放一帧静态的图像数据，以支持智能屏接口的自刷新功能；另外该模组上还集成了背光灯控制电路，可以调整PWM的占空比与脉冲频率，来控制 LCD面板显示的亮度。

15.2.2　CPU侧显示驱动模块

通过前面硬件部分的讲述， CPU侧关于显示相关的驱动应该牵涉到 3个部分：一是 LCD控制器图像数据输出部分；二是 SPI LCD模组显示控制部分；三是 PWM LCD模组的背光控制部分。在软件上，该三部分功能都被适配成了独立模块，其中 LCD控制器图像输出是显示子系统中驱动最主要组成。

首先， LCD控制器作为挂载在 PXA910CPU总线上的一个模块，是Platform虚拟总线上的一个设备。因此，在我们的驱动里应该将该驱动注册到 Platform bus。同时，我们根据第二篇所学知识，容易理解该驱动应该是一个字符类驱动。代码如下：

```
static int __init pxafb_init(void)
{
    ...

    return platform_driver_register(&pxafb_driver);
}
```

由于该模块实现的是系统图像输出基础功能，该模块一定要被编译入内核的。系统启动会去执行 pxafb_init（）函数，进而导致该函数最后一条语句的执行，它会去注册定义的 pxafb驱动。根据 PNP驱动程序的架构，其又必定导致相应 probe函数的执行。对应的 probe驱动程序如代码清单 15-8所示。

代码清单 15-8　pxafb_probe（）

```
static int __devinit pxafb_probe(struct platform_device *dev)
{
    ...
    fbi = NULL;
    ...
    ret = pxafb_parse_options(&dev->dev, g_options);
    ...
    fbi = pxafb_init_fbinfo(&dev->dev);
    ...

    fbi->backlight_power = inf->pxafb_backlight_power;
    fbi->lcd_power = inf->pxafb_lcd_power;

    r = platform_get_resource(dev, IORESOURCE_MEM, 0);
    ...
    r = request_mem_region(r->start, resource_size(r), dev-
>name);
    ...
    fbi->mmio_base = ioremap(r->start, resource_size(r));
```

```
    ...
    fbi->dma_buff_size = PAGE_ALIGN(sizeof(struct
pxafb_dma_buff));
    fbi->dma_buff = dma_alloc_coherent(fbi->dev, fbi-
>dma_buff_size,
                &fbi->dma_buff_phys, GFP_KERNEL);
    ...
    ret = pxafb_init_video_memory(fbi);
    ...
    irq = platform_get_irq(dev, 0);
    ...
    ret = request_irq(irq, pxafb_handle_irq, IRQF_DISABLED,
"LCD", fbi);
    ...
    ret = pxafb_smart_init(fbi);
    ...
    ret = register_framebuffer(&fbi->fb);
    ...

    pxafb_overlay_init(fbi);
...
    /*经过上面的准备,下面使能LCD控制器*/
    set_ctrlr_state(fbi, C_ENABLE);
...
}
```

在该 probe函数里，将完成访问该 LCD设备所需要显存、中断等资源的分配。这里，我们应特别关注函数 register_framebuffer（）的调用，因为它将向 Framebuffer子系统注册一个 Framebuffer设备 fb0。其实该设备正对应着前面所讲三个图层中的第一个图层。而紧接着调用的 pxafb_overlay_init（），就将完成对应第二个图层的设备 fb1的注册。

至于 SPI控制器以及 PWM控制器的驱动，也是字符类驱动，但它们在这里从属于 Framebuffer显示子系统。这些驱动由于属于 CPU侧，在 CPU原厂的 SDK包里一般也会带有它们的源码，也并不复杂，参照第二篇所述相信大家很容易掌握它们。而且一般读者仅是运用这些驱动，基于这些驱动的修改或重写的机会不多，这里就不做更多的讲解了。

15.2.3　LCM驱动模块

LCM就是 LCD模组，是系统中的基础显示器件。它的驱动就是设置与控制该显示器件，使得系统中的其他程序可以通过该驱动顺利地对该显示器件进行输出操作。这将是显示驱动开发工程师没法避免的工作之一。为了让大家更清楚地理解 PXA910 LCD显示驱动，下面给出实际的T35KPS05E LCM的接口图（见图 15-7），后面驱动参考代码中很大部分就是对 LCM所用这些接口管脚的 GPIO配置工作。

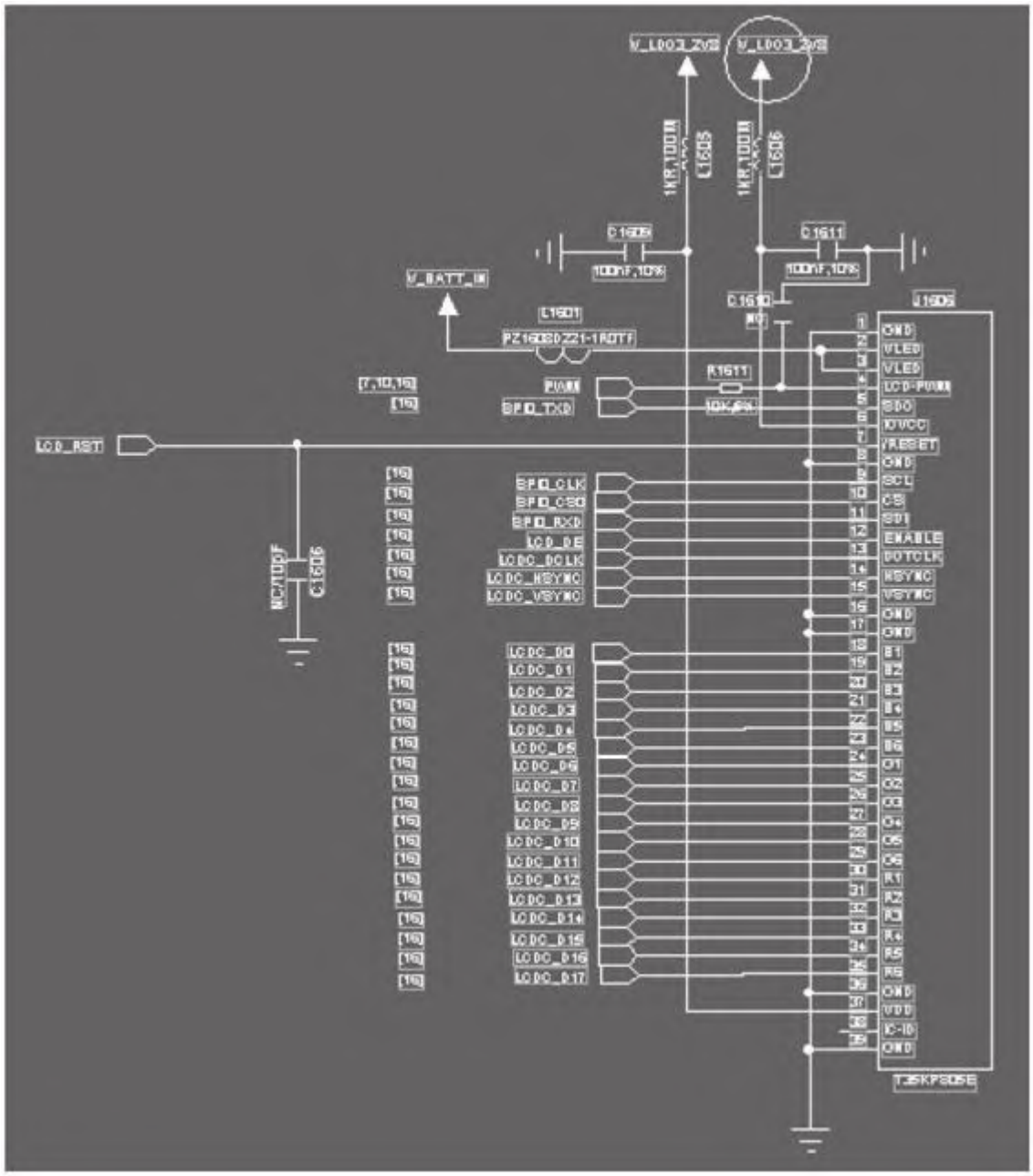


图15-7　LCD模组显示相关原理图

事实上，即使配置并使能了 LCM， LCD控制器工作正常，输出了图像数据，但用户有可能还是从 LCM看不到任何东西，因为 LCM还有一个背光器件。只有 LCM背光提供光源， LCM上的 TFT才能控制液晶，显示用户需要的图像。因此，显示驱动开发工程师还需要有一个背光模块要开发。

LCM的显示控制有两项主要工作：一个是配置 RGB666并行总线，连接 CPU的 pin脚；另一个就是通过 SPI总线，接收来自 CPU的写寄存器命令，由于这些寄存器相关位的改变，进而导致 LCM TFT控制矩阵工作模式的改变。参见代码清单 15-9。

代码清单 15-9　LCM上电 /断电控制

```
static int lead_lcd_power(struct pxa168fb_info *fbi, unsigned
int spi_gpio_cs, unsigned int spi_gpio_reset, int on)
{
    ...
    mfp_config(ARRAY_AND_SIZE(lead_lcd_pin_config));//用于配置
LCM要用到的CPU GPIO管脚
    if (on) {    //启动LCM
        if(!lcm_powered) {
            ...

            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data1,
ARRAY_SIZE(lead_spi_cmdon_data1), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data1_1,
ARRAY_SIZE(lead_spi_cmdon_data1), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data1_2,
ARRAY_SIZE(lead_spi_cmdon_data1), 8);
            ...

            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2_1,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2_2,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
```

```
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2_3,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2_4,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data2_5,
ARRAY_SIZE(lead_spi_cmdon_data2), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data3,
ARRAY_SIZE(lead_spi_cmdon_data3), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdon_data4,
ARRAY_SIZE(lead_spi_cmdon_data4), 8);
            ...
        }
        else {
            err = gpio_spi_send_leadlcd(lead_spi_cmdoff_data1,
ARRAY_SIZE(lead_spi_cmdoff_data1), 8);
            ...
            err = gpio_spi_send_leadlcd(lead_spi_cmdoff_data2,
ARRAY_SIZE(lead_spi_cmdoff_data2), 8);
            ...
        }

    }
else {
        err = gpio_spi_send_leadlcd(lead_spi_cmdoff_data1,
ARRAY_SIZE(lead_spi_cmdoff_data1), 8);
        ...
        err = gpio_spi_send_leadlcd(lead_spi_cmdoff_data2,
ARRAY_SIZE(lead_spi_cmdoff_data2), 8);
        ...
        err = gpio_request(spi_gpio_reset,
"LEAD_LCD_SPI_RESET");
        ...
    }
    ...
}
```

在上面的代码中，把通过 SPI写 LCM寄存器的功能代码封装在函数 gpio_spi_send_leadlcd（），这样就可以把具备某一项控制功能的寄存器组放在一起配置，这样方便根据 LCM的功能项调整相应的代码段。比如当我们调整 gamma数组值时，使得红、绿、蓝各色系，在人的视觉感受能有线性的变化。该函数代码参见清单 15-10。

代码清单 15-10　gpio_spi_send_leadlcd（）

```
static int gpio_spi_send_leadlcd(u8 *dat, u32 size, u32
bit_len)
{
    ...
    unsigned int cs_gpio = mfp_to_gpio(MFP_PIN_GPIO107);
    int err = gpio_request(cs_gpio, "LCD CS");
    ...
    gpio_direction_output(cs_gpio, 1);
    udelay(20);

    clk_gpio = mfp_to_gpio(MFP_PIN_GPIO108);
    err = gpio_request(clk_gpio, "LCD CLK");
    ...

    d_gpio = mfp_to_gpio(MFP_PIN_GPIO104);
    err = gpio_request(d_gpio, "LCD data");
    ...
    gpio_direction_output(cs_gpio, 0);
    gpio_direction_output(d_gpio, 0);    //用来告知接下来数据线上发
送的是SPI配置命令
    gpio_direction_output(clk_gpio, 0);
    msleep(10);
    gpio_direction_output(clk_gpio, 1);
    while(bit_cnt){
        bit_cnt--;
        if((*val >> bit_cnt) & 1){
            gpio_direction_output(d_gpio, 1);
        }else{
            gpio_direction_output(d_gpio, 0);
        }
        udelay(20);
        gpio_direction_output(clk_gpio, 0);
        udelay(20);
        gpio_direction_output(clk_gpio, 1);
        udelay(20);
    }
    val++;

    // Send data next
    datasize = size-1;
    for(; datasize; datasize--){
        bit_cnt = bit_len;
        gpio_direction_output(cs_gpio, 0);
```

```
        gpio_direction_output(d_gpio, 1);     //用来告知接下来数据
线上发送的数据
        gpio_direction_output(clk_gpio, 0);
        msleep(10);
        gpio_direction_output(clk_gpio, 1);
        udelay(20);
        while(bit_cnt){
            bit_cnt--;
            if((*val >> bit_cnt) & 1){
                gpio_direction_output(d_gpio, 1);
            }else{
                gpio_direction_output(d_gpio, 0);
            }
            udelay(20);
            gpio_direction_output(clk_gpio, 0);
            udelay(20);
            gpio_direction_output(clk_gpio, 1);
            udelay(20);
        }
        val++;
        udelay(20);
        gpio_direction_output(cs_gpio, 1);

    }
    gpio_free(cs_gpio);
    gpio_free(clk_gpio);
    gpio_free(d_gpio);
    return 0;
}
```

细心的读者也许已发现，函数 gpio_spi_send_leadlcd（）根本就没有使用 SPI控制器，而是使用 GPIO引脚仿真了一个 SPI协议来实现 CPU对 LCM的写操作。这也是我们前面并不讲 CPU侧 SPI控制器驱动模块的原因之一。因为很多外设模组，虽然声称也使用 SPI接口，但在 SPI协议上或时序上有所更改，以标识其独特性。此时，使用一般的 SPI控制器，通过标准的 SPI协议与这些模组交互就不会成功。但作为驱动开发工程师，不能就此束手，而可以像上面代码一样，用 3～ 4根 GPIO引脚，参照模组规范说明，仿真出其所需要的 SPI协议，在 CPU与模组之间搭建起一道交互的桥梁。

至于 LCM的背光控制，在软件上也是一个独立的模块，其驱动也是一个独立的字符类驱动。关于它的驱动架构这里就不再讨论，而是关注它设置背光亮度强弱的功能函数，参见代码清单 15-11所示。

代码清单 15-11　LCM的背光控制

```
static int pwm_backlight_update_status(struct backlight_device
*bl)
{
    struct pwm_bl_data *pb = dev_get_drvdata(&bl->dev);
    int brightness = bl->props.brightness;
    int max = bl->props.max_brightness;

    if (bl->props.power != FB_BLANK_UNBLANK)
        brightness = 0;

    if (bl->props.fb_blank != FB_BLANK_UNBLANK)
        brightness = 0;

    if (pb->notify)
        brightness = pb->notify(pb->dev, brightness);

    if (brightness == 0) {
        pwm_config(pb->pwm, 0, pb->period);
        pwm_disable(pb->pwm);
    } else {
        pwm_config(pb->pwm, brightness * pb->period / max, pb-
>period);
        pwm_enable(pb->pwm);
    }
    return 0;
}
```

该函数的主要功能就是接收来自上层用户或其他程序对背光亮度级别的设置，根据该设置值，配置相应的 PWM脉冲频率与占空比（其实就是使用前面所讲的 CPU侧 PWM控制器的驱动模块），进而控制背光LED的亮度，从而使得 LCM的亮度按用户设置或环境变化。

15.3　Android系统对 Framebuffer的使用

Android上层应用要想使用 Framebuffer子系统硬件，也是符合第 14章所讲的架构的。当然，该子系统也有其自身的特色，参见图 15-8。

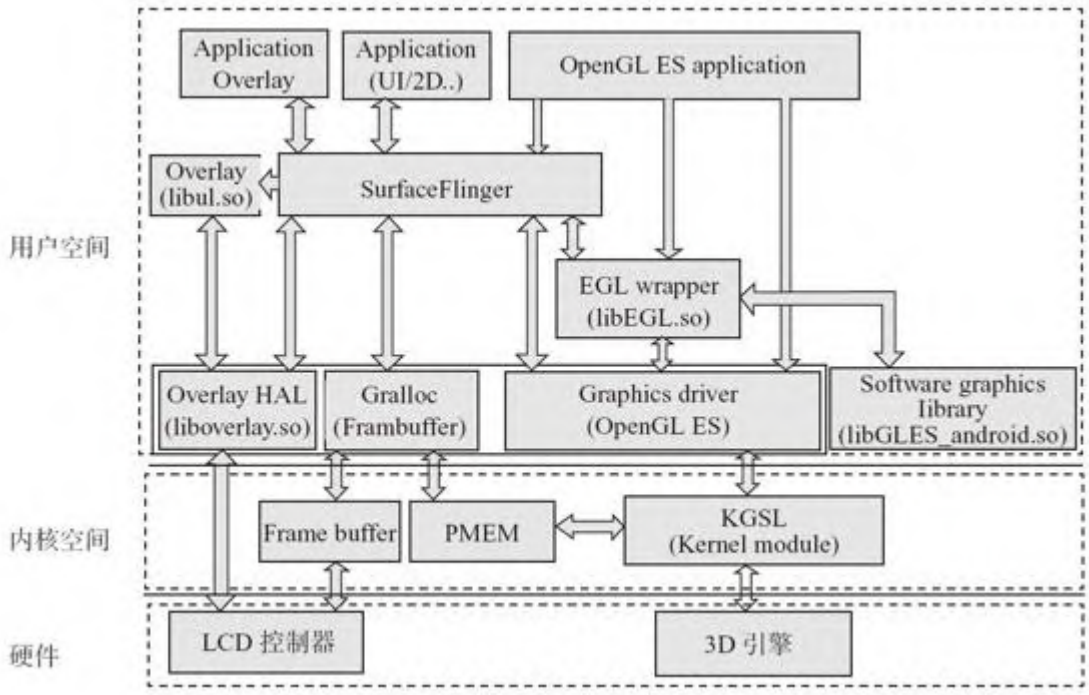


图15-8　Android显示子系统架构

如图 15-8所示，显示子系统的 HAL层有三个模板。其中 Gralloc模块是显示子系统的主模块，其负责往 Frame buffer显存传送数据；Graphics driver模块是 3D显示 HAL模块，其负责调用 3D引擎计算处理 3D图像数据（这块由相应厂商提供库，没有源码，根据 Apache License，这是许可的）；另一个 Overlay HAL模块负责视频动态图像数据处理。

在 Android Framebuffer显示子系统中，有一个核心组件 SurfaceFlinger。它是显示子系统的 Service进程，它将各种应用程序的 2D、 3D surface进行组合、合并，最终得到的一个 main surface数据送入显存。简单地说， surfaceFlinger就像是画布，它不关心画上去的内容，只是一味地执行合成功能，当然要根据画的位置、大小以及效果等参数。这很像 Photoshop中的各个图层，你可以

在不同的图层画任意的内容，每个图层可以设置位置、大小、效果参数等，最终通过 merge功能合成一个图层。

从应用的角度看，每个应用程序可以有一个或多个图形界面，每个界面可以看作是一个 surface。首先每个 surface有它的位置、大小、内容等元素，这些元素是可以随便变化的；另外不同的 surface的位置会有重叠，会涉及透明度等效果处理问题，这些都是通过 surfaceFlinger来完成的。当然， surfaceFlinger只是担任一个组织协调的职责，最终功能的落实要依靠 Frame buffer显示驱动以及 LCD控制器等硬件上，但前提是 surfaceFlinger需要把相关参数计算好，如重叠的位置等。

作为开发人员来讲，我觉得还应了解 Android显示子系统的 Client/Server架构。服务端（即 SurfaceFlinger）负责 surface的合成等处理工作，客户端根据 SurfaceFlinger所提供的接口给上层操作各自的 surface，并向服务端发送消息完成实际显示工作。服务端主要由 C++代码编写而成。客户端代码分为两部分，一部分是由 Java提供的供应用程序使用的 API，另一部分则是由 C++编写的底层实现。如图 15-9所示。

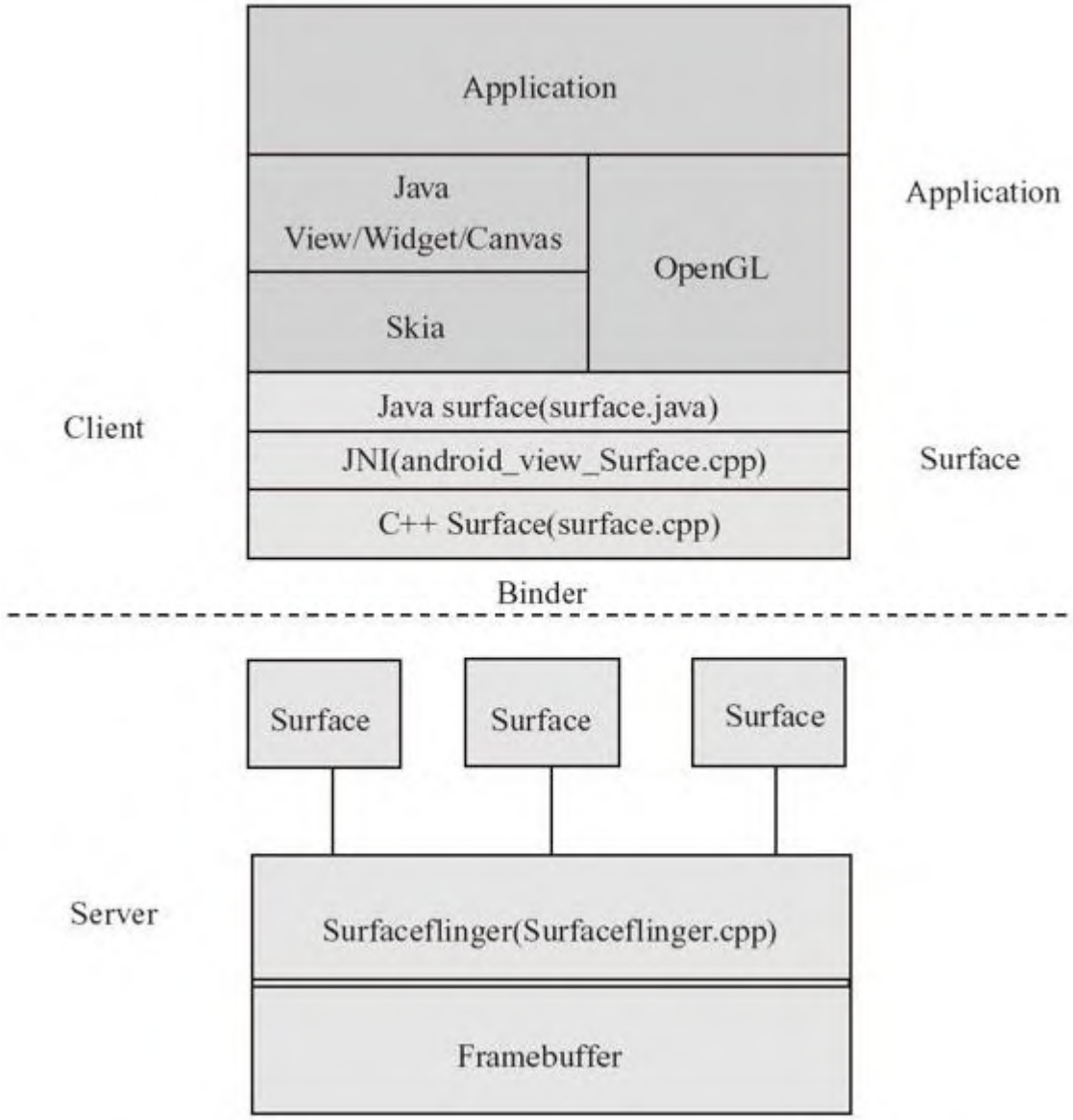


图15-9　Android的Client/Server架构

通过图 15-9可以进一步看清楚， Android系统是如何对 Framebuffer显示子系统（包括 FrameBuffer驱动）进行调用的。

另外从图 15-9中看到有一个 Binder组件。在第 14章我们已知道它是Android进程间通信机制，其底层技术是 Android专有的 Binder驱动。在这里，它负责调用显示功能的客户进程与 Surfaceflinger服务进程之间的通信。

第16章　Input子系统

介绍完输出子系统，我们再来讲讲Android的输入子系统。

现在的计算机设备都要有相应的输入设备来辅助人机交互，像Android智能机中的键盘、触摸屏等，就是这类输入设备。下面我们开始讲述Android的Input子系统。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

16.1　Linux Input一般子系统

在真正讲述 Android的 Input子系统之前，我们先看看 Linux一般 Input子系统。

Input子系统处理输入事务，输入设备的驱动程序通过 Input输入子系统提供的接口注册到内核，利用子系统提供的功能与用户空间或系统中的其他程序交互。整个 Input子系统实现可看作三层：输入驱动层（ input driver）、输入核心层（ input core），以及事件处理层（ event handler）。图 16-1是 Linux Input子系统的框架。

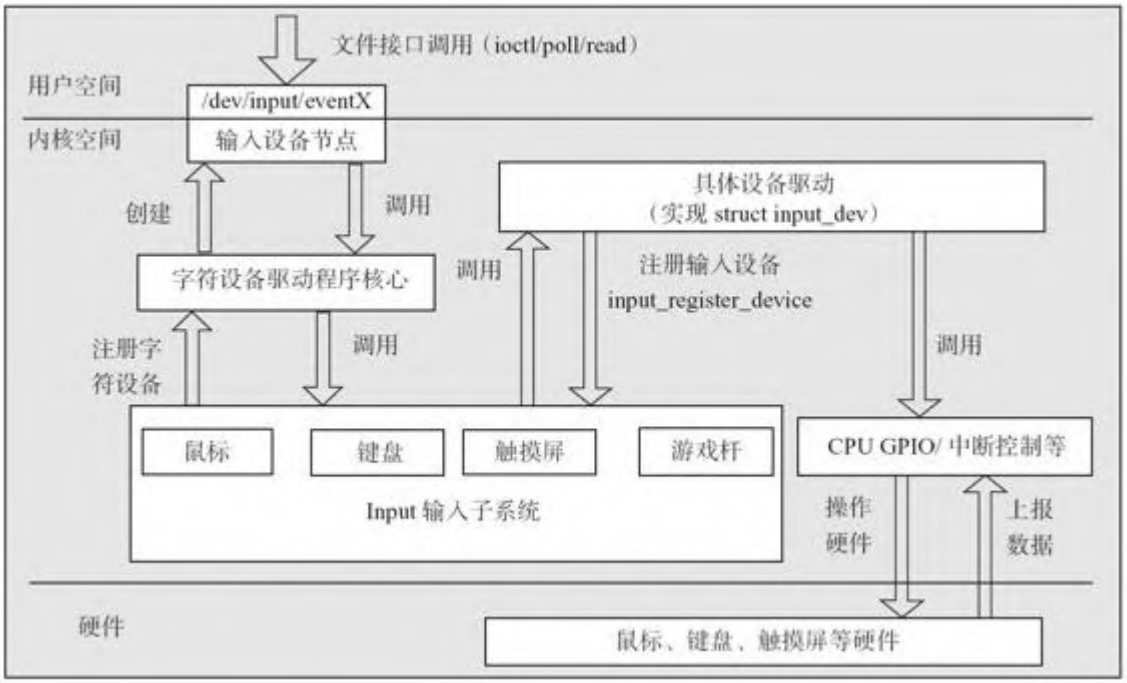


图16-1　Linux Input子系统框架

Input子系统三层结构中输入驱动层，就是针对各个具体输入设备的驱动功能程序；输入核心层则是 Linux内核为了统一管理各输入设备及其驱动的模块；而事件处理层用来将底层输入硬件事件，翻译成为上层用户空间所认知的软事件。从图 16-1中我们还看到用户空间对输入设备的访问，是通过“字符设备驱动程序核心”转入的，这就告诉我们一般的输入设备都是字符类设备。

16.1.1　Input数据结构

1.input_dev

从图 16-1中可以看到， input_dev是 Input子系统中最重要的数据结构。它代表了一类输入设备。每个具体的 input驱动程序，都必须分配和初始化这样的一个结构体。由于该结构体中成员较多，我们就不列出完整代码清单了，而是挑选其中核心关键成员来讲，希望使大家对该数据结构有一个更清晰脉络的理解。

（ 1）若干个数组成员

```
unsigned long evbit[BITS_TO_LONGS(EV_CNT)];    //事件支持的类型
// 下面是每种类型支持的编码
unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; //按键
unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; //绝对坐标,其中触摸屏驱动使用的就是这个
unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
```

evbit[BITS_TO_LONGS（ EV_CNT）]这个数组以位掩码的形式代表了这个设备支持的事件的类型。设置方式如：

```
dev->evbit[0]=BIT（EV_SYN）|BIT（EV_KEY）|BIT（EV_ABS）
```

absbit[BITS_TO_LONGS（ ABS_CNT）]这个数组也是以位掩码的形式代表这个类型的事件支持的编码。触摸屏驱动支持 EV_ABS，所以要设置这个数组，有一个专门设置这个数组的函数 input_set_abs_params，如代码清单 16-1。

代码清单 16-1　input_set_abs_params（）

```
static inline void input_set_abs_params(struct input_dev *dev,
int axis, int min, int max, int fuzz, int flat)
{
    dev->absmin[axis] = min;
    dev->absmax[axis] = max;
    dev->absfuzz[axis] = fuzz;
    dev->absflat[axis] = flat;

    dev->absbit[BIT_WORD(axis)] |= BIT_MASK(axis);  //填充了
absbit这个数组
}
```

触摸屏驱动中是这样调用的：

```
input_set_abs_params(dev, ABS_X, 0, 0x3FF, 0, 0);     //这个是设
置ad转换的x坐标
input_set_abs_params(dev, ABS_Y, 0, 0x3FF, 0, 0);    //这个是设置
ad转换的y坐标
input_set_abs_params(dev, ABS_PRESSURE, 0, 1, 0, 0); //这个是设置
触摸屏是否按下的标志
```

其中，表示设置 ABS_X/ABS_Y编码值范围为 0～ 0x3ff。而这个值的范围是由 CPU或其他硬件的 ADC限制所致，因为这些 ADC的位数为 10位，所以不会超过 0x3ff。

（ 2） struct input_id id成员

这个成员用来标识设备驱动特征。

```
struct input_id {
    __u16 bustype;   //总线类型
    __u16 vendor;    //生产厂商
    __u16 product;   //产品类型
    __u16 version;   //版本
};
```

如果需要特定的事件处理器来处理这个设备的话，这几个就非常重要，因为 Input子系统核心要通过它们将设备驱动与事件处理层联系起来的。但是因为输入设备驱动所用的事件处理器一般为通用型，则这个初始化也无关紧要。

（ 3）其他成员

还有其他一些成员也比较重要，都是由 Input子系统核心来处理的，驱动程序可以不用管。

2.input_handler

它是事件处理器的数据结构，代表一个事件处理器。

（ 1）几个操作函数指针

```
void (*event)(struct input_handle *handle, unsigned int type,
unsigned int code, int value);
int (*connect)(struct input_handler *handler, struct input_dev
*dev, const struct input_device_id *id);
void (*disconnect)(struct input_handle *handle);
void (*start)(struct input_handle *handle);
```

event函数是当事件处理器接收到来自 input设备传来的事件时调用的处理函数，负责处理事件，这个函数非常重要，在下一小节会详细分析。

connect函数是当一个 input设备模块注册到内核的时候调用的，将事件处理器与输入设备联系起来，也就是将 input_dev和 input_handler配对的函数。

disconnect函数实现与 connect函数相反的功能。

start函数用来启动给定的 input handler。该函数一般由 input核心层在调用 input handler的 connect（）函数之后被调用。

（ 2）两个 id数组

```
const struct input_device_id *id_table; //这个是事件处理器所支持的input设备
const struct input_device_id *blacklist; //这个是事件处理器应该忽略的input设备
```

这两个数组都会用在 connect函数中， input_device_id结构与input_id结构类似，但是 input_device_id多了一个 flag成员，该成员设定了某些输入设备的属性，如 busytype、 vendor等，用来判断事件处理器要与何种输入设备相关联。

（ 3）两个链表

```
struct list_headh_list;  //这个链表用来链接它所支持的input_handle结构,input_dev与input_handler配对之后就会生成一个input_handle结构
struct list_headnode;    //链接到input_handler_list,这个链表链接了所有注册到内核的事件处理器
```

3.input_handle

该结构体代表一个成功配对的 input_dev和 input_handler。参见代码清单 16-2。

代码清单 16-2　input_handle

```
struct input_handle {
    void *private;   //每个配对的事件处理器都会分配一个对应的设备结构,如evdev事件处理器的evdev结构,注意这个结构与设备驱动层的input_dev不同,初始化handle时,保存到这里
    int open;        //打开标志,每个input_handle 打开后才能操作,这个一般通过事件处理器的open方法间接设置
    const char *name;
    struct input_dev *dev;  //关联的input_dev结构
```

```
    struct input_handler *handler; //关联的input_handler结构
    struct list_head    d_node;  //input_handle通过d_node连接到
input_dev的h_list链表上
    struct list_head    h_node;  //input_handle通过h_node连接到
input_handler的h_list链表上
};
```

总之， input_dev就是硬件驱动层，代表一个 input设备。input_handler是事件处理层，代表一个事件处理器。 input_handle则属于 Input子系统核心层，代表一个配对的 input设备与 input事件处理器。 input_dev通过全局的 input_dev_list链接在一起，设备注册的时候实现这个操作。 input_handler通过全局的input_handler_list链接在一起，事件处理器注册的时候实现这个操作。 input_handle没有一个全局的链表，它注册的时候将自己分别挂在了 input_dev和 input_handler的链表上，通过 input_dev和input_handler就可以找到 input_handle。在设备注册和事件处理器注册的时候，都要进行配对工作，配对后就会实现链接。通过input_handle也可以找到 input_dev和 input_handler。

16.1.2 Input内核模块

因为用户需要的输入方式多种多样， Linux Input子系统所能支持的输入设备，可以说是千差万别。为了读者对千差万别的 input子系统有更清楚的认识，我们再来讲讲 Input子系统的三层内核模块。先看看图 16-2。

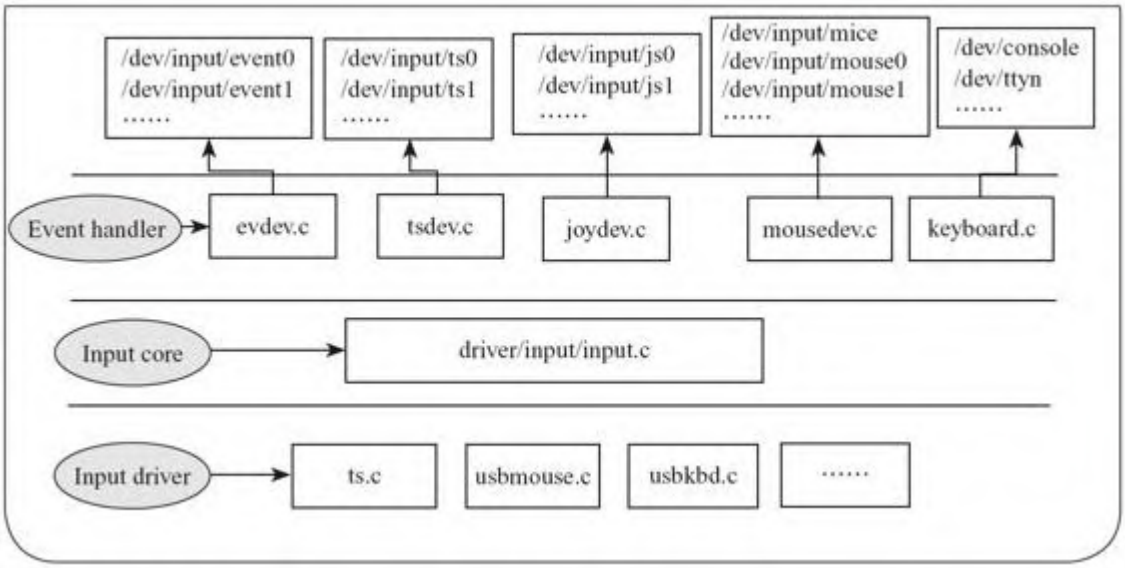


图16-2　Input三层内核模块

在图 16-2的三层模块中， Input内核模块主要完成 3件事：一是完成形形色色 input设备在 Linux内核的注册，二是完成事件处理器 input_handler在 Linux内核的注册；三是完成 Linux input事件上报。

1.input设备的注册

前面讲过，在 linux驱动中描述一个 input设备可以用 input_dev来表示。在编写一个具体 input设备驱动时，就需要为你的 input设备先分配这样一个结构体，并初始化 dev结构体。这部分功能由函数 input_allocate_device（）来完成，参见代码清单 16-3。

代码清单 16-3　input_allocate_device（）

```
struct input_dev *input_allocate_device(void)
{
    struct input_dev *dev;

    dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);
    if (dev) {
        dev->dev.type = &input_dev_type;
        dev->dev.class = &input_class;
        device_initialize(&dev->dev);
```

```
        mutex_init(&dev->mutex);
        spin_lock_init(&dev->event_lock);
        INIT_LIST_HEAD(&dev->h_list);
        INIT_LIST_HEAD(&dev->node);

        __module_get(THIS_MODULE);
    }
    return dev;
}
```

分配 input_dev结构体所要的内存，并将 dev的成员赋值初始化后，然后就可以调用内核 API函数 input_register_device（）来向input子系统注册，参见代码清单 16-4。

代码清单 16-4　input_register_device（）

```
int input_register_device(struct input_dev *dev)
{
    ...
    error = device_add(&dev->dev);
    ...
    list_add_tail(&dev->node, &input_dev_list);
    ...
    list_for_each_entry(handler, &input_handler_list, node)
    input_attach_handler(dev, handler);
    ...
}
```

从上面的代码可看到， input设备的注册还是与所有的 linux设备注册一样调用 device_add将设备注册为 linux设备。同时通过 list_add_tail将设备添加到 linux内核全局链表 input_dev_list。然后通过 list_for_each_entry为设备找到属于自己的 handler。

2.input_handler的注册

一般来说 input_handler的注册会在 input_dev之前，常见的 input_handler有： mousedev handler（处理来自鼠标类的 input事件）、 joydev_handler（处理游戏摇杆类事件）、 kdev_handler（处理来自键盘类事件）、 evdev_handler（响应绝大

部分的事件，默认的 input处理事件）。 Input_handler的注册由API函数 input_register_handler完成，参见代码清单 16-5。

代码清单 16-5　input_register_handler（）

```
int input_register_handler(struct input_handler *handler)
{
    ...
    INIT_LIST_HEAD(&handler->h_list);

    if (handler->fops != NULL) {
        if (input_table[handler->minor >> 5]) {
            retval = -EBUSY;
            goto out;
        }
        input_table[handler->minor >> 5] = handler;
    }
    ...
    list_add_tail(&handler->node, &input_handler_list);
    list_for_each_entry(dev, &input_dev_list, node)
    input_attach_handler(dev, handler);
    ...
}
```

函数先初始化 input handler在 linux的内核链表，然后如果handler的文件操作（ fops）不为空就会为这个 handler分配一个次设备号， handler->minor>>5即 handler通过 32为倍数来区分的，换句话说每个 handler处理的 input event不能超过 32。现在可以看到无论是设备的注册还是 handler注册都会调用input_attach_handler，该函数的代码参见代码清单 16-6。

代码清单 16-6　input_attach_handler（）

```
static int input_attach_handler(struct input_dev *dev, struct
input_handler *handler)
{
...
id = input_match_device(handler, dev);
...
```

```
error = handler->connect(handler, dev, id);
...
}
```

每个 handler在注册的时候都有自己的 id_table，如果设备与 input handler能够匹配成功的话，就会调用 input handler的 connect函数。在 input_match_device中先会进行 input device的 id.bustype、 id.vendor、 id.product和 id.version等的匹配，然后会去匹配 evbit、 keybit、 relbit、 absbit等，参见代码清单 16-7。

代码清单 16-7　input_match_device（）

```
static const struct input_device_id *input_match_device(struct
input_handler *handler, struct input_dev *dev)
{
    ...
    for (id = handler->id_table; id->flags || id->driver_info;
id++) {

        if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
            if (id->bustype != dev->id.bustype)
                continue;
        ...
        MATCH_BIT(evbit,  EV_MAX);
        ...
        if (!handler->match || handler->match(handler, dev))
            return id;
    }
}
```

如果 handler和 device能够匹配的话，就会调用 handler的 match回调函数，但是在 linux input Handler中不是所有的 handler都实现了这个回调函数。目前只有 joydev handler实现了。再看看通用事件处理器 evdev handler的输入设备 id匹配表，参见代码清单 16-8。

代码清单 16-8　evdev_ids

```
static const struct input_device_id evdev_ids[] = {
    { .driver_info = 1 },    //可匹配所有输入设备驱动
    { },                     //空元素标识该表的结束
};
```

可以看到 evdev是作为一个通用的 handler，可匹配所有输入设备，去处理各 input device的事件，也就是说一旦有设备注册就会去调用evdev的 connect函数，参见代码清单 16-9。

代码清单 16-9　evdev_connect（）

```
static int evdev_connect(struct input_handler *handler, struct
input_dev *dev  const struct input_device_id *id)
{
    ...
    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
    ...
    evdev->handle.dev = input_get_device(dev);
    evdev->handle.name = dev_name(&evdev->dev)
    evdev->handle.handler = handler;
    evdev->handle.private = evdev;
    evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE +
minor);
    evdev->dev.class = &input_class;
    evdev->dev.parent = &dev->dev;
    evdev->dev.release = evdev_free;
    device_initialize(&evdev->dev);

    error = input_register_handle(&evdev->handle);
    ...
    error = device_add(&evdev->dev);
    ...
}
```

在上面的代码后端，会调用 device_add（）等函数实施设备注册，以向 Linux系统新创建一个 evdev设备。该设备的文件节点为/dev/input/eventX。上层应用可直接通过这些设备文件来获取底层的输入设备事件。 evdev_connect（）函数还会调用

Input_register_handle，将匹配好的 input设备和 input handler分别加到自己的 input设备链表和 handler链表中。

3.input事件上报

一般 input事件在底层上报都是通过中断方式，如最常用的按键或者触摸屏就是采用中断方式。这里我们以一个具体的触摸屏的驱动为例，来分析一下事件的上报过程。

```
input_report_abs(dev, ABS_X, MTOUCH_GET_XC(mtouch->data));
input_report_abs(dev, ABS_Y, MTOUCH_MAX_YC -
MTOUCH_GET_YC(mtouch->data));
input_report_key(dev, BTN_TOUCH, MTOUCH_GET_TOUCHED(mtouch-
>data));
input_sync(dev)
```

上面就是一个具体触摸屏接收到中断信号，调用中断处理函数来处理事件时所调用的方法来完成一次上报事件，它以 input_sync来表示一次完成的事件上报，而这个函数都会统一调用 input_event来进行处理，参见代码清单 16-10。

代码清单 16-10　input_event（）

```
void input_event(struct input_dev *dev, unsigned int type,
unsigned int code, int value)
{
    ...
    input_handle_event(dev, type, code, value);
    ...
    }
static void input_handle_event(struct input_dev *dev, unsigned
int type, unsigned int code, int value)
{
    ...
    int disposition = INPUT_IGNORE_EVENT;
    ...
    if (disposition & INPUT_PASS_TO_HANDLERS)
        input_pass_event(dev, type, code, value);
```

```
    ...
}
```

在 input_handle_event（）函数中会有一个 disposition变量，来判定事件是交由谁来处理，有的交给 device来处理，有的交给 handler处理，还有的则交给 device和 handler共同处理。一般情况都是交给handler来处理。也就是说又会调用 input_pass_event（）函数来继续处理，参见代码清单 16-11。

代码清单 16-11　input_pass_event（）

```
static void input_pass_event(struct input_dev *dev, unsigned
int type, unsigned int code, int value)
{
    ...
    handle = rcu_dereference(dev->grab);
    if (handle)
        handle->handler->event(handle, type, code, value);
    ...
}
```

在之前的 handler或者 device注册的时候，每当一个 device和 handler发生匹配都会注册一个 handle，然后将 handler和 device都加到自己链表中，这时候就可以通过 handle来调用 handler的 event函数，产生相应的 input事件。而这些 input事件会被存放在指定的buffer中，由上层读取并做出相应的响应。

16.2　Android Input子系统实践

Android Input子系统继承了 Linux的 Input子系统，下面我们就通过Android中的具体输入设备来看看 Android基于 Input子系统的实践与应用。

以下示例基于 Marvell PXA910CPU、 AW9523A扩展 GPIO，实现支持可达 64个按键的 Keypad。

16.2.1　硬件基础

在当前的嵌入式 CPU中，由于所支持的功能越来越多，而它们的资源有限，特别是 GPIO引脚尤其显得不足。当想将 Android系统运用于一些行业领域，比如物流行业的数字数据采集终端，就会需要一个小的全键盘。为此我们引入一个 GPIO扩展硬件模块 AW9523A，以实现7×8矩阵按键 input设备。其硬件原理图如图 16-3所示。

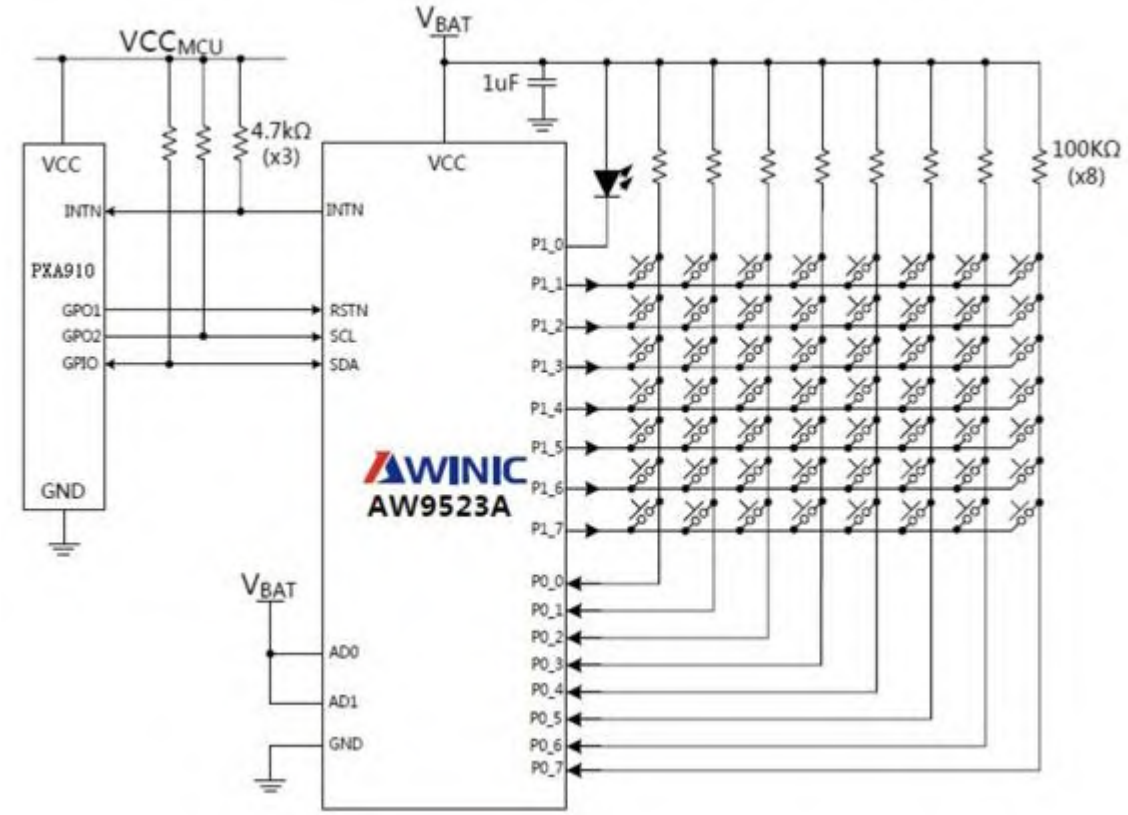


图16-3　7×8矩阵小键盘

AW9523A通过 I2C总线，与 PXA910 CPU相连。当键盘矩阵上有按键按下或松开时， AW9523A会给 CPU上报中断。 CPU接收中断后，在中断处理中可以通过 I2C总线读取 AW9523A的寄存器，以获得是哪个按键在动作以及动作的类型等信息。当然，这些信息离不开一个键盘驱动的支持。

16.2.2　Input驱动模块

这里要描述的是一个具体的 Input驱动模块。它就是 AW9523A扩展键盘驱动。

首先，在该驱动模块的初始化函数中，我们调用了 i2c_add_driver（）来向 I2C总线挂载该驱动，参见代码清单 16-12。

代码清单 16-12　aw9523a_i2c_init（）

```
static int aw9523a_i2c_init(void)
{
    ...
    retval = i2c_add_driver(&aw9523a_driver);
     ...
}
```

而在数据结构 aw9523a_driver中，会指定该驱动模块加载时的 probe函数： aw9523a_probe（）。该 probe函数的代码骨架如代码清单 16-13所示。

代码清单 16-13　aw9523a_probe（）

```
static int __devinit aw9523a_probe(struct i2c_client *client,
const struct i2c_device_id *id)
{
    struct aw9523a_data *aw9523a;
```

```
    struct input_dev *input;
    ...
    struct i2c_adapter *adapter = to_i2c_adapter(client-
>dev.parent);
    ...
    /* Chip is valid and active. Allocate structure */
    aw9523a = kzalloc(sizeof(struct aw9523a_data), GFP_KERNEL);
    input = input_allocate_device();
    ...
    dev_set_drvdata(&client->dev, aw9523a);
...
    aw9523a->client = client;
    aw9523a->input = input;

    INIT_DELAYED_WORK(&aw9523a->dwork, aw9523_worker);
    spin_lock_init(&aw9523a->lock);
...
    input->name = ?aw9523a?;
    input->id.bustype = BUS_I2C;

    input->keycode = aw9523a->keycodes;
    input->keycodesize = sizeof(aw9523a->keycodes[0]);
    input->keycodemax = ARRAY_SIZE(array56_key);

    __set_bit(EV_KEY, input->evbit);
    __clear_bit(EV_REP, input->evbit);

    for (i = 0; i < ARRAY_SIZE(array56_key); i++) {
        aw9523a->keycodes[i] = array56_key[i];
        __set_bit(array56_key[i], input->keybit);
    }
    __clear_bit(KEY_RESERVED, input->keybit);

    input_set_drvdata(input, aw9523a);
    error = input_register_device(aw9523a->input);
    ...

    error = request_irq(client->irq, aw9523a_irq,
IRQF_TRIGGER_FALLING, ?aw9523a?, aw9523a);
    ...

    i2c_set_clientdata(client, aw9523a);

    this_client = client;
    input_dev = input;

#ifdef CONFIG_HAS_EARLYSUSPEND
    aw9523a_early_suspend.suspend = aw9523a_suspend;
```

```
    aw9523a_early_suspend.resume = aw9523a_resume;
    aw9523a_early_suspend.level =
EARLY_SUSPEND_LEVEL_DISABLE_FB+3;
    register_early_suspend(&aw9523a_early_suspend);
#endif
    ...
}
```

在该函数中调用 input_register_device（），完成该键盘输入设备在 Android Input内核子系统的注册；同时申请了一个降沿中断，以接收来自 AW9523A的中断请求。中断处理见代码清单 16-14。

代码清单 16-14　aw9523a_irq（）

```
static irqreturn_t aw9523a_irq(int irq, void *_aw9523a)
{
    struct aw9523a_data *aw9523a = _aw9523a;
    ...
    spin_lock_irqsave(&aw9523a->lock, flags);
    ...
    schedule_delayed_work(&aw9523a->dwork,
msecs_to_jiffies(0));

    spin_unlock_irqrestore(&aw9523a->lock, flags);
    return IRQ_HANDLED;
}
```

在该中断处理例程中，会去调度在 probe函数中定义的 worker线程：aw9523a_worker。在该线程的主体函数中，将完成真正的按键信息收集、转换与上报工作。参见代码清单 16-15。

代码清单 16-15　aw9523a_worker（）

```
static void aw9523_worker(struct work_struct *work)
{
    struct aw9523a_data *aw9523a = _aw9523a;

    aw9523_p0_int_disable(aw9523a->client);
```

```
    ...
    KeyBoard_Key = keyboard_get_press_key(aw9523a);    //从
AW9523A获得按键消息

    //响应按键释放消息
    if (KeyBoardKey_State == KEY_STATE_NULL ||
KeyBoardKey_State == KEY_STATE_RELEASED)
    {
        if (KeyBoard_Key != 0xFF) {
            KeyBoardKey_State = KEY_STATE_PRESSED;
            KeyBoard_Key_Previous = KeyBoard_Key;
            ...

            // Backspace Key Proc
            if(KeyBoard_Key_Previous == KEY_BACKSPACE)
            {
                Key_Irq_Enter_Flag = TRUE;
                Key_Long_Press_Flag = TRUE;

                input_report_key(input_dev,
KeyBoard_Key_Previous, 1);
                input_sync(input_dev);
                input_report_key(input_dev,
KeyBoard_Key_Previous, 0);
                input_sync(input_dev);

                schedule_delayed_work(&aw9523a->dwork,
msecs_to_jiffies(BACKSPACE_KEY_LONG_PRESS_FIRST_TIME));
            }
            ...
            goto irq_proc_exit;
        }
        else  // KeyBoard_Key == 0xFF
        {
              //按键值为255,第二次按键up进入
            goto irq_proc_exit;
        {
    }
    else if (KeyBoardKey_State == KEY_STATE_PRESSED)
    {
        //第二次按键down进入,且不是常按按键的情况
        if(KeyBoard_Key != 255 && Key_Long_Press_Flag == FALSE)
        {
            goto irq_proc_exit;
        }
        //第二次按键down进入,且不是常按按键的情况
        if(KeyBoard_Key != 255 && Key_Long_Press_Flag == TRUE)
```

```
        {
            goto Num_Key_Long_Press_Proc;
        }

        if (KeyBoard_Key != KeyBoard_Key_Previous) {
            KeyBoardKey_State = KEY_STATE_RELEASED;
            ...

            //if key is SHIFT
            if(KeyBoard_Key_Previous == KEY_FUNC_SWITCH)
            {
                SHIFT_KEY_PRESS_FLAG = TRUE;
                Current_Select_Multi_Key_Pos = 1;

                Start_Time_Dealy_Flag = FALSE;
                Key_DealyTime_Flag = FALSE;

                Input_Char_Select_Status = FALSE;
                SHIFT_FUNC_FIRST_TIME = TRUE;
            }
            // end if

            //UP Key Proc,增加Vol+
            if(KeyBoard_Key_Previous == KEY_UP)
            {
                if((PHONE_status == 1)||(isInMusicScreen == 1))
                {
                    input_report_key(input_dev,  KEY_VOLUMEUP,
1);
                    input_sync(input_dev);
                    input_report_key(input_dev,  KEY_VOLUMEUP,
0);
                    input_sync(input_dev);

                    KeyBoard_Key_Previous_Realse =
KeyBoard_Key_Previous;

                    goto irq_proc_exit;
                }
            }

            //Down Key Proc, 增加Vol-
            if(KeyBoard_Key_Previous == KEY_DOWN)
            {
                if((PHONE_status == 1)||(isInMusicScreen == 1))
                {
                    input_report_key(input_dev,
```

```
KEY_VOLUMEDOWN, 1);
                    input_sync(input_dev);
                    input_report_key(input_dev,
KEY_VOLUMEDOWN, 0);
                    input_sync(input_dev);

                    KeyBoard_Key_Previous_Realse =
KeyBoard_Key_Previous;

                    goto irq_proc_exit;
                }
            }

            // Backspace Key & Left Key & Right Key Proc &&
(PHONE_status != 1)
            if((KeyBoard_Key_Previous == KEY_BACKSPACE) ||
(KeyBoard_Key_Previous == KEY_UP) || (KeyBoard_Key_Previous ==
KEY_DOWN))
            {
                KeyBoard_Key_Previous_Realse =
KeyBoard_Key_Previous;
                goto irq_proc_exit;
            }
            ...

            KeyBoard_Key_Previous_Realse =
KeyBoard_Key_Previous;

            input_report_key(input_dev,  KeyBoard_Key_Previous,
1);
            input_sync(input_dev);
            input_report_key(input_dev,  KeyBoard_Key_Previous,
0);
            input_sync(input_dev);
             ...
        }
        else
        {
                //按键长按处理分支
Num_Key_Long_Press_Proc:
            ...
            // Backspace Key Proc
            if(KeyBoard_Key_Previous == KEY_BACKSPACE)
            {
                input_report_key(input_dev,
KeyBoard_Key_Previous, 1);
                input_sync(input_dev);
```

```
                input_report_key(input_dev,
KeyBoard_Key_Previous, 0);
                input_sync(input_dev);

                Key_Irq_Enter_Flag = TRUE;
                schedule_delayed_work(&aw9523a->dwork,
msecs_to_jiffies (BACKSPACE_KEY_LONG_PRESS_TIME));
            }
            ...
            KeyBoardKey_State = KEY_STATE_PRESSED;
        }
    }
irq_proc_exit:
    aw9523_i2c_write(aw9523a->client, 0x05, P1_VALUE);
    udelay(5);
    aw9523_get_p0(aw9523a->client);
    udelay(5);
    aw9523_get_p1(aw9523a->client);
    udelay(5);
    aw9523_p0_p1_interrupt_setting(aw9523a->client);
    ...
    return 0;
}
```

该按键处理线程的核心工作就是从 AW9523A获取按键消息，并将按键消息转换成合适的按键事件，通过调用 input_sync（）将该事件上报。

16.3　Android系统对 Input的使用

Android系统对 Input（输入）子系统的集成与使用也沿袭 Android的HAL架构，以便上层应用对底层的 Input内核模块进行调用，对 Input事件进行响应。 Android输入子系统的基本架构图如图 16-4所示。

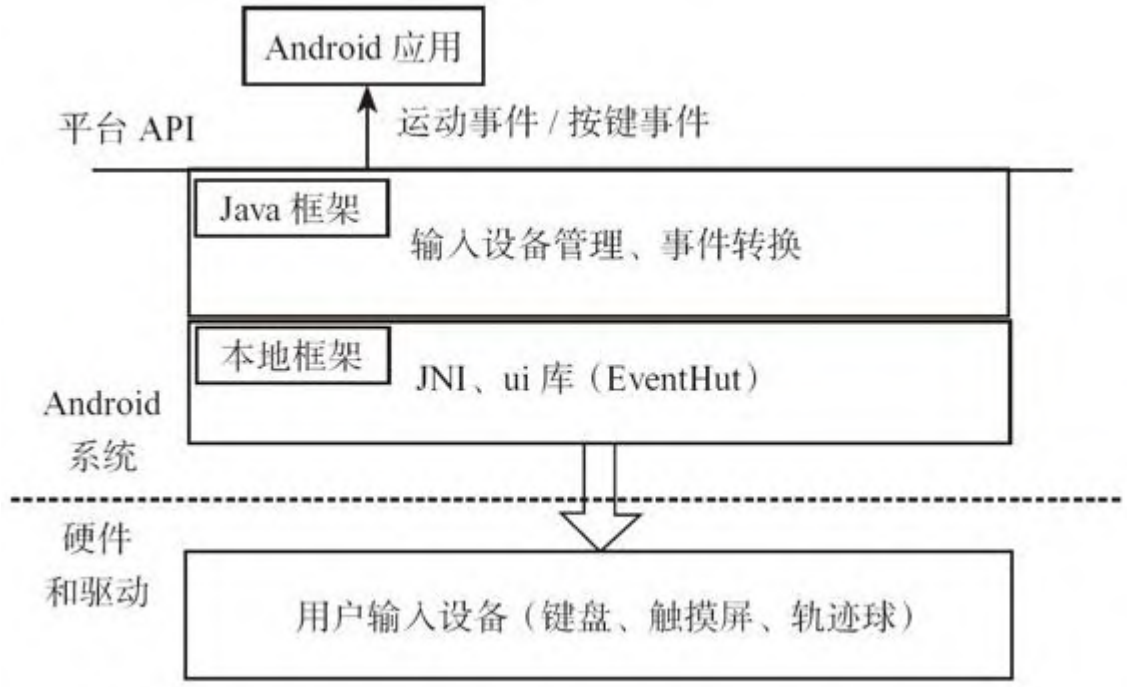


图16-4　Android输入子系统架构图

图 16-4中所显示的 EventHub就是 Android输入子系统的 HAL层，它负责与底层 Input子系统内核各模块（包括上面开发的 AW9523A扩展keypad驱动）的交互。事实上， Android中 EventHub没有独立的库，它是 libui的一部分。 EventHub在初始化时会调用功能函数scanDeviceLocked（），来扫描打开底层 Input子系统中的输入设备。

Android输入子系统也没有独立的服务线程，它通过系统的WindowManagerService向上层应用提供相应的输入服务。其实负责Android input子系统的 InputManager就是 WindowManagerService的有机组成部分之一。 InputManager会通过 EventHub去读取 Input事件，并分发这些事件。 EventHub会用到 inboundqueue和

outboundqueue两个事件队列处理按键等输入事件，而驱动中input_sync（）就是这两个队列元素生成源。 EventHub会启动InputDispatcherThread线程来处理输入事件：先是从 inboundqueue中取出新的 input event处理，处理完成再生成另一新的 input event放进 outboundqueue；接着从 outboundqueue中取新的 input event，处理完成则生成新的 inputmessage；最后调用 ::send（）函数向上层应用发送 inputmessage。总之 EventHub要负责完成从input event到 input message的转化。 Android Input子系统的事件传递流程如图 16-5所示。

图 16-5中还用虚线框出了工程师开发工作的重点：最底下的部分就是诸如 AW9523A驱动之类；而右边部分的 kl文件和 kcm文件分别用于上层动态定义按键布局和按键功能转换映射。

另外，提醒读者注意的是，在 Android系统中，为了加快按键等输入事件的响应，在 WindowManagerService进程与 Android其他进程之间的通信没有采用 Binder方式，而是采用了共享内存 +管道的 IPC通信方式。其中管道用来告诉其他进程有新事件到达，以及其他进程已完成事件的处理。而共享内存则用来存放相应的事件信息。因为这不是本书重点，更详细的知识大家可以查询相关的资料。

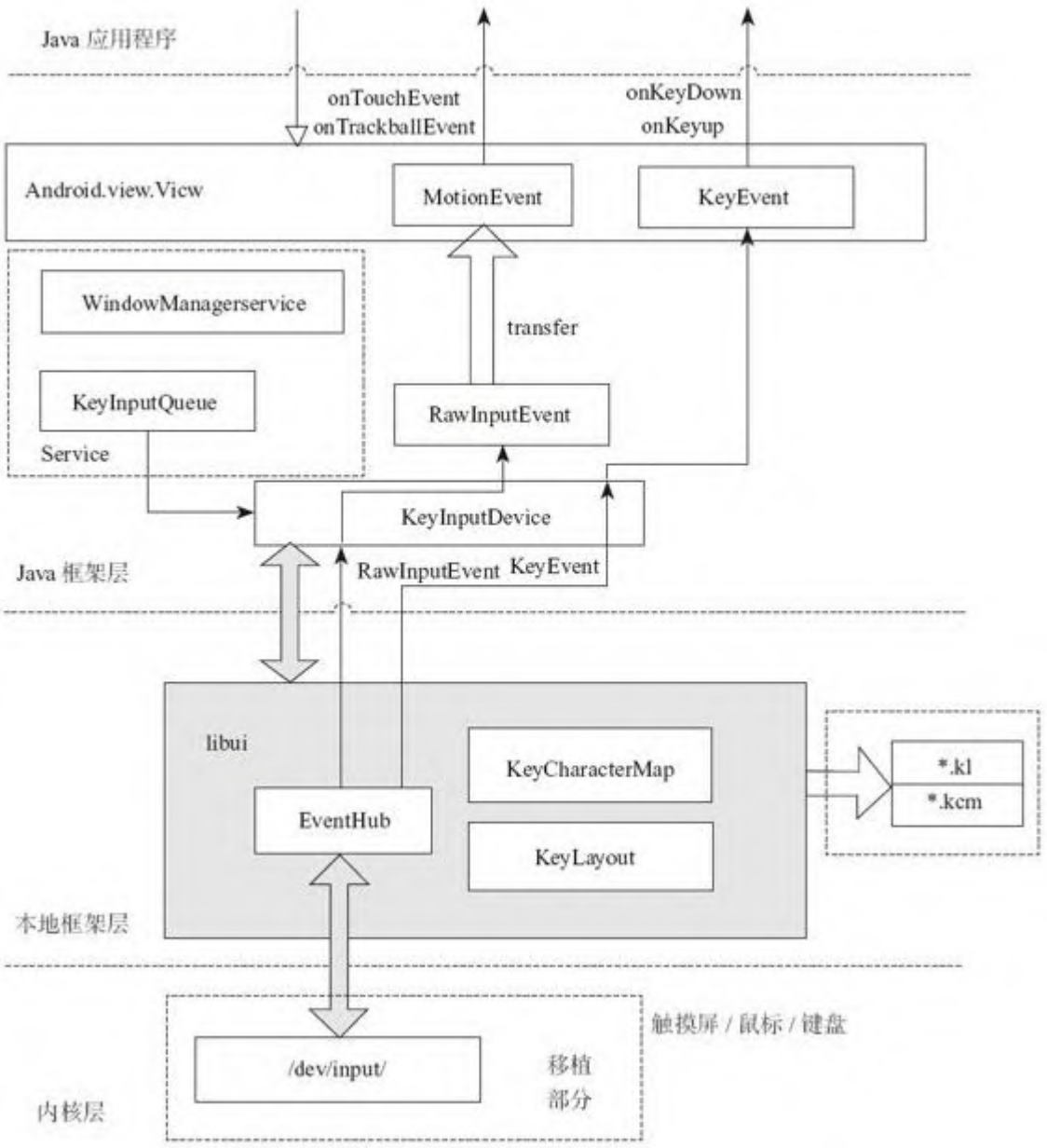


图16-5　Android输入事件传递

第17章　V4L2子系统

前面两章分别介绍Android的一个输出子系统和一个输入子系统。本章我们再介绍既是输入又是输出的子系统——V4L2子系统。

V4L2是V4L的升级版本，是Linux为视频设备程序开发提供的一套接口规范。像我们Android手机中常有的Camera正是这类视频设备，它负责视频图像的采集与输出。

该接口规范包括了一套数据结构，以及访问底层V4L2驱动的接口。

注：V4L2提供的接口有许多，但驱动并不需要实现所有的接口功能；而应根据具体情况选择相应的接口，实现我们需要的功能。

在Android系统中，针对Camera等视频设备也遵循V4L2规范。

17.1　Linux V4L2一般子系统

V4L，其全称是 Video4Linux（即 Video for Linux），是 Linux内核中关于视频设备的 API接口，涉及开关视频设备，以及从该类设备采集并处理相关的音、视频信息。 V4L从 Linux2.1版本的内核中开始出现。

现在 Linux内核中用的是 V4L2，即 Video4Linux2（即 Video for Linux Two），其是修改 V4L相关 Bug后的一个升级版，始于 Linux 2.5内核。

V4L2的主设备号是 81，次设备号为 0～ 255；这些次设备号里又分为多类设备：视频设备、 Radio（收音机）设备、 Teletext on VBI 等。因此 V4L2设备对应的文件节点有： /dev/videoX、 /dev/vbiX、/dev/radioX。

其中 VBI设备是基于电视场消隐实现远程传送文字的技术与设备，我们一般不会接触到。对于 Radio设备，即用于收发声音。但要提醒注意的是，对于声音的采集与处理，在我们的 Android手持设备中常会有个 Mic设备，它则是属于 ALSA子系统的。因此我们主要讨论的是 Video设备。

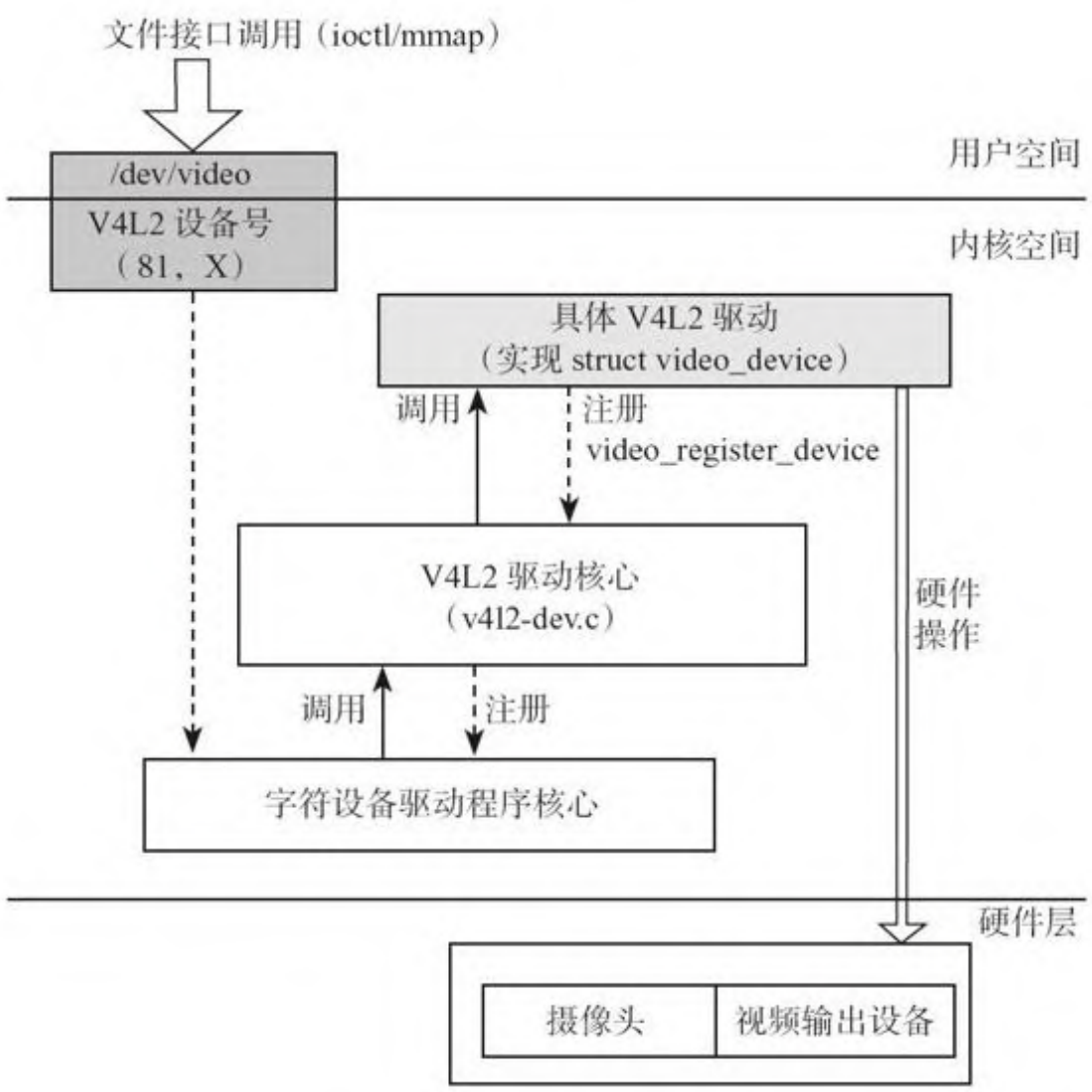


图17-1　V4L2框架

图 17-1是基于 Video设备的 V4L2框架。

17.1.1　V4L2数据结构

V4L2常用的数据结构体定义在 include/linux/videodev2.h中。参见代码清单 17-1。

代码清单 17-1　videodev2.h中定义的 V4L2数据结构

```
struct v4l2_requestbuffers        //申请帧缓冲,对应命令
VIDIOC_REQBUFS
struct v4l2_capability        //视频设备的功能,对应命令
VIDIOC_QUERYCAP
struct v4l2_input               //视频输入信息,对应命令
VIDIOC_ENUMINPUT
struct v4l2_standard        //视频制式所采用标准,比如PAL、NTSC,对应命
令VIDIOC_ENUMSTD
struct v4l2_format               //帧的格式,对应命令VIDIOC_G_FMT、
VIDIOC_S_FMT等
struct v4l2_buffer               //驱动中的一帧图像缓存,对应命令
VIDIOC_QUERYBUF
struct v4l2_crop              //视频信号矩形边框
v4l2_std_id               //视频制式标准ID号
```

当然，上面的基础数据结构会牵涉到更多的数据结构，例如： struct v4l2_format就会用到 struct v4l2_pix_format，参见代码清单 17-2。

代码清单 17-2　v4l2_format

```
struct v4l2_format
{
    enum v4l2_buf_type type; // 数据流类型,必须永远是
V4L2_BUF_TYPE_VIDEO_CAPTURE
    union
    {
        struct v4l2_pix_format    pix;
        struct v4l2_window        win;
        struct v4l2_vbi_format    vbi;
        __u8    raw_data[200];
    } fmt;
};
struct v4l2_pix_format
{
    __u32                    width;        // 宽,必须是16的倍数
    __u32                    height;       // 高,必须是16的倍数
    __u32                    pixelformat; // 视频数据存储类型,例
如是YUV4：2：2还是RGB888
    enum v4l2_field             field;
```

```
    __u32                bytesperline;
    __u32                       sizeimage;
    enum v4l2_colorspace        colorspace;
    __u32                       priv;
};
```

在上述数据结构中， struct v4l2_requestbuffers用来申请存放视频采集图像所要的帧缓冲，而 struct v4l2_buffer则用来指定与描述一帧帧缓冲。参见代码清单 17-3。

代码清单 17-3　帧缓冲相关数据结构

```
struct v4l2_requestbuffers
{
    __u32               count;  //缓存数量,也就是说在缓存队列里保持多少张照片
    enum v4l2_buf_type  type;  //数据流类型,对于视频捕获设备则是 V4L2_BUF_TYPE_VIDEO_CAPTURE
    enum v4l2_memory    memory; // V4L2_MEMORY_MMAP 或 V4L2_MEMORY_USERPTR
    __u32               reserved[2];
};
struct v4l2_buffer {
    __u32            index;
    enum v4l2_buf_type      type;
    __u32            bytesused;
    __u32            flags;
    enum v4l2_field          field;
    struct timeval          timestamp;
    struct v4l2_timecode    timecode;
    __u32            sequence;

    /* memory location */
    enum v4l2_memory        memory;
    union {
        __u32           offset;
        unsigned long   userptr;
    } m;
    __u32            length;
    __u32            input;
    __u32            reserved;
};
```

17.1.2　V4L2接口

从所支持的设备来讲， V4L2有 5类接口：

1）视频采集接口（ video capture interface），也就是以高频头或Camera为输入源，驱动该类设备，接收相应的视频信息并处理。

2）视频输出接口（ video output interface），也就是以高频头为输出源，驱动该类设备发射之类的电视信号。

3）直接传输视频接口（ video overlay interface），也就是从Camera等输入源进行数据采集后，不做处理，直接输出到高频头等输出源上；可以理解为这样的 Camera设备有输入端口，也有输出端口；事实上 Android拍照应用在进行预览时，可能就是这种模式。

4）视频间隔消隐信号接口（ VBI interface），这个前面我们有解释了。

5）收音机接口（ radio interface），也就是以 AM或 FM高频头为输入源，驱动该类设备接收相应的音频流。

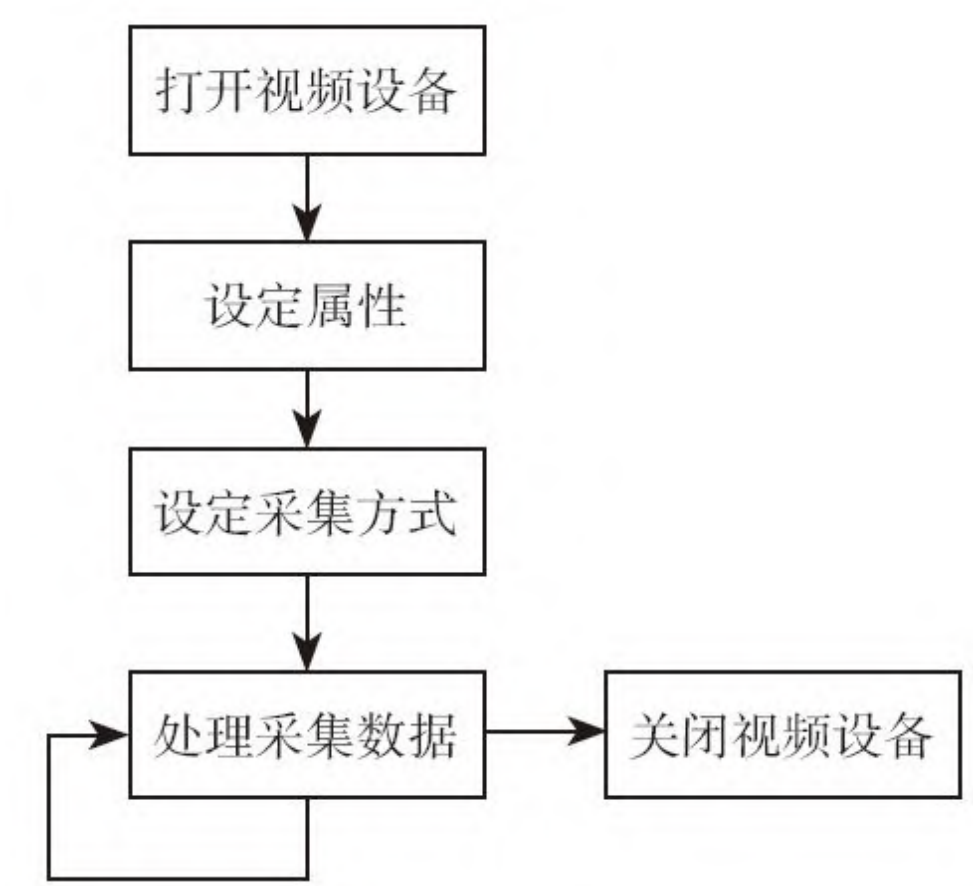


图17-2　视频采集流程

V4L2具体的驱动就是让程序可以找到相应的设备，并操作该设备。我们这里更应该关注的地方是，具体驱动是如何通过接口函数实现具体功能，以及响应上层程序的功能接口。下面我们以上面提到的视频采集接口为例来看看 V4L2的实现，以及需要向上层提供哪些具体功能接口。视频采集流程见图 17-2。

V4L2中定义了许多 IOCTL命令以供上层调用，这些 IOCTL命令将会被各接口使用，它们的定义也在文件 include/linux/videodev2.h中。参见代码清单 17-4。

代码清单 17-4　V4L2常用 IOCTL

```
VIDIOC_REQBUFS        //分配内存
VIDIOC_QUERYBUF       //把VIDIOC_REQBUFS中分配的数据缓存转换成物理地址
```

```
VIDIOC_QUERYCAP      //查询驱动功能
VIDIOC_ENUM_FMT      //获取当前驱动支持的视频格式
VIDIOC_S_FMT         //设置当前驱动的视频捕获格式
VIDIOC_G_FMT         //读取当前驱动的视频捕获格式
VIDIOC_TRY_FMT       //验证当前驱动的显示格式
VIDIOC_CROPCAP       //查询驱动的修剪能力
VIDIOC_S_CROP        //设置视频信号的矩形边框
VIDIOC_G_CROP        //读取视频信号的矩形边框
VIDIOC_QBUF          //把数据从缓存中读取出来
VIDIOC_DQBUF         //把数据放回缓存队列
VIDIOC_STREAMON      //开始视频显示函数
VIDIOC_STREAMOFF     //结束视频显示函数
VIDIOC_QUERYSTD      //检查当前视频设备支持的标准,例如PAL或NTSC
```

从图 17-2可以分析出， V4L2针对视频采集需要以下接口：

1）打开设备文件。

视频设备与其他设备一样被看作一个文件，所以也是用 open这个接口函数来打开该设备。通常，视频设备的打开有两种方式：

```
// 用非阻塞模式打开摄像头设备
int cameraFd;
cameraFd = open("/dev/video0", O_RDWR | O_NONBLOCK, 0);
// 如果用阻塞模式打开摄像头设备,上述代码变为:
//cameraFd = open("/dev/video0", O_RDWR, 0);
```

上层应用程序如果使用非阻塞模式打开该设备，则即使还没有捕获到视频信息，该设备的驱动也要把缓存（ DQBUFF）里的内容返回给上层应用程序。

2）取得设备所能提供的功能（ Capability）。

我们打开了视频设备就会想到要使用该设备。但在使用该设备的具体功能之前，应当了解该设备支持哪些 Capability：可读、可写、可对视频信号进行何种调制、支持 VBI等。这些 Capability的获得则是通过 IOCTL接口来调用的。

```
// 取得设备的capability
struct v4l2_capability capability;
int ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
```

3）取得与设定视频所支持的制式。

在不同地区往往所采用的视频制式不一样（就电视而言，我国是 PAL，而欧洲许多国家则是 NTSC）。我们应先了解视频设备所支持的视频制式，然后再根据制式要求设定想要的更具体的视频制式。它们也是通过 IOCTL接口来调用。

```
// 获得设备所支持的视频制式
v4l2_std_id std;
do {
    ret = ioctl(fd, VIDIOC_QUERYSTD, &std);
} while (ret == -1 && errno == EAGAIN);
switch (std) {
        case V4L2_STD_NTSC:
            //...
        case V4L2_STD_PAL:
            //...
}

// 设置设备的视频输出制式
struct v4l2_format    fmt;
memset ( &fmt, 0, sizeof(fmt) );
fmt.type                = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width       = 720;
fmt.fmt.pix.height      = 576;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
fmt.fmt.pix.field        = V4L2_FIELD_INTERLACED;
if (ioctl(fd, VIDIOC_S_FMT, &fmt) == -1) {
  return -1;
}
```

4）向驱动申请帧缓存。

我们采集到的视频信息应该有存放的地方，在 V4L2中这些存储空间被称为帧缓存。它也是通过 IOCTL接口来调用。

```
// 申请内存
struct v4l2_requestbuffers req;
if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
    return -1;
}
```

在上一小节中，我们了解到 v4l2_requestbuffers结构中定义了缓存的数量，驱动会据此申请对应数量的视频缓存。多个缓存可以用于建立先入先出缓冲（ FIFO），以提高视频采集的效率。

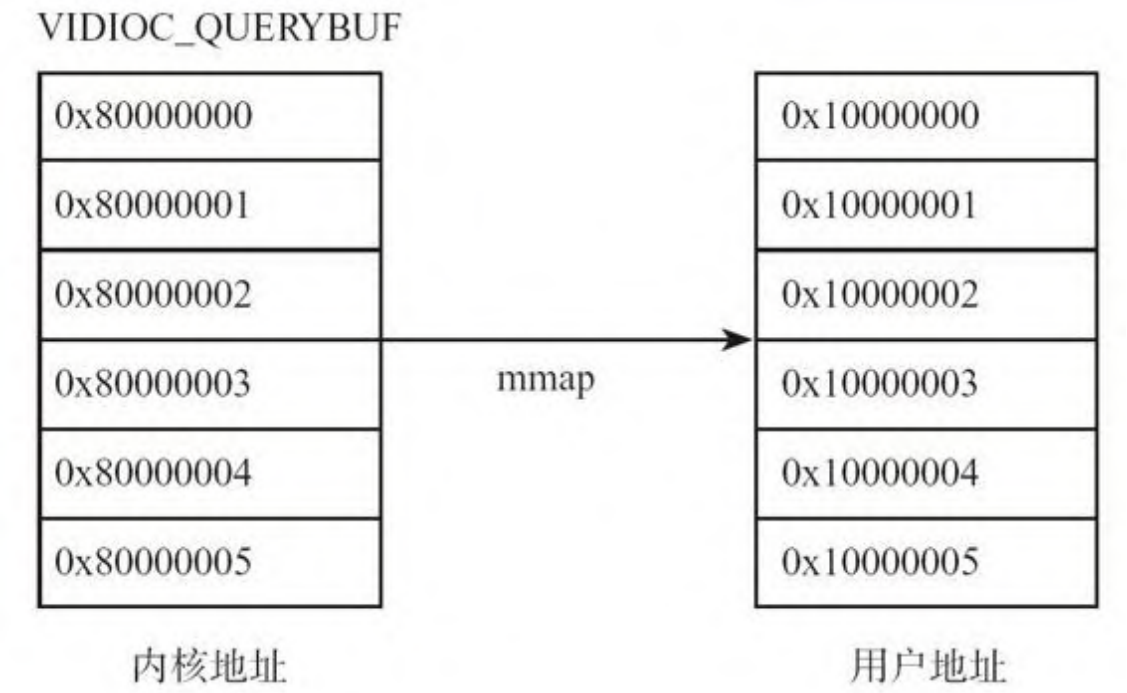


图17-3　视频帧缓冲mmap示意图

5）获取每个缓存的信息并映射到用户空间。

接着，我们就应调用 IOCTL命令 VIDIOC_QUERYBUF来获取某个帧缓存的地址，然后使用 mmap（）函数将其转换成上层应用程序中的绝对地址，最后把这个帧缓存置于缓存队列中（见图 17-3）。

下面则是相应的接口调用示例：

```
typedef struct VideoBuffer {
    void   *start;
    size_t  length;
} VideoBuffer;

VideoBuffer* buffers = calloc( req.count, sizeof(*buffers) );
struct v4l2_buffer buf;        //声明一视频帧

// 对帧缓存中的每个视频帧做好映射
for (numBufs = 0; numBufs < req.count; numBufs++) {
    memset( &buf, 0, sizeof(buf) );
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = numBufs;
    // 读取某缓存帧的信息
    if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {
        return -1;
    }

    buffers[numBufs].length = buf.length;
    // 转换成绝对地址
    buffers[numBufs].start = mmap(NULL, buf.length,
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        fd, buf.m.offset);

    if (buffers[numBufs].start == MAP_FAILED) {
        return -1;
    }

    // 放入缓存队列
    if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
        return -1;
    }
}
```

6）启动视频采集。

准备好存储空间，下面就应开始捕获视频数据了。但在启动之前，我们有必要先了解上层应用如何获取底层 V4L2驱动捕获的数据。

前面我们讲过， Linux OS把系统使用的内存划分成用户空间和内核空间，分别由应用程序管理和操作系统管理。应用程序可以直接访问用户空间内存的地址，而内核空间存放的是供内核访问的代码和数据，用户不能直接访问。而 V4L2捕获的数据最初是存放在内核空间的，这意味着用户不能直接访问该段内存，必须通过某些手段来转换地址。

一共有三种视频采集方式：

1）使用 read、 write方式（ read-write）：直接使用 read和 write接口函数进行读写。这种方式最简单，但是这种方式会在用户空间和内核空间不断复制数据，同时在用户空间和内核空间占用了大量内存，效率不高。

2）内存映射方式（ mmap）：把设备里的内存映射到应用程序中的内存空间，直接处理设备内存，这是一种有效的方式。上面讲过的 mmap函数正是使用这种方式。

3）用户指针模式（ userptr）：内存由用户空间的应用程序分配，并把地址传递到内核中的驱动程序，然后由 V4L2驱动程序直接将数据填充到用户空间的内存中。这点需要在 v4l2_requestbuffers里将 memory字段设置成 V4L2_MEMORY_USERPTR。

第一种方式效率是最低的，后面两种方法都能提高执行的效率，但是对于 mmap方式，由于 buffer是在内核空间中分配的，这种情况下这些 buffer不能被交换到 SWAP中。虽然这种方法不怎么影响读写效率，但是它一直占用着内核空间中的内存，当系统的内存有限的时候，如果同时运行大量的进程，则对系统的整体性能有一定的影响。

所以，对于以上三种视频数据收集方式的选择，我们推荐的顺序是 userptr、 mmap、 read-write。当使用 mmap或 userptr方式的时候，有一个环形缓冲队列的概念，这个队列中，有 n个 buffer，驱动程序采集到的视频帧数据就存储在每个 buffer中。在每次用 VIDIOC_DQBUF取出一个 buffer并且处理完数据后，一定要用 VIDIOC_QBUF将这个 buffer再次放回到环形缓冲队列中。环形缓冲队列也使得这两种视频采集方式的效率高于 read-write方式。

下面的 IOCTL的 VIDIOC_STREAMON命令则用来启动视频的采集：

```
int buf_type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
int ret = ioctl(fd, VIDIOC_STREAMON, &buf_type);
```

7）取出 FIFO缓存中已采样的帧缓存，将刚处理完的缓存置入缓存队列尾部，以便视频采集过程可循环使用它们。

针对前面提到的环形缓冲队列，数据的缓存基于 FIFO方式，下面是基本的工作代码，它用到了两个 IOCTL命令： VIDIOC_DQBUF和 VIDIOC_QBUF。

```
struct v4l2_buffer buf;
memset(&buf,0,sizeof(buf));
buf.type=V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory=V4L2_MEMORY_MMAP;
buf.index=0;
//读取缓存
if (ioctl(cameraFd, VIDIOC_DQBUF, &buf) == -1)
{
    return -1;
}
//.............视频处理算法
//重新放入缓存队列
if (ioctl(cameraFd, VIDIOC_QBUF, &buf) == -1) {

    return -1;
}
```

根据我们关于 Linux OS知识的了解，像部分采样数据等烦琐工作，最好是放在一个独立的线程中来完成。

8）停止视频采集。

不再需要进行视频采集的话，则可以通过下面的 IOCTL命令来停止：

```
int ret=ioctl（fd，VIDIOC_STREAMOFF，&buf_type）；
```

9）关闭相应设备。

不再需要使用该设备，则可以通过 close接口函数来关闭相应的设备：

```
close（fd）;
```

当然，上面这些是基本接口，针对具体的设备可能需要我们实现更多的接口，比如，当一个视频设备有多路视频源时，就还应该增加一个视频源选择接口。

至于除视频采集之外其他类的接口，我们通过其工作流程的分析也不难找到它们所应实现的接口。若想全面地了解 V4L2，不妨查看相应的规范文档《 videoforlinux2》。

17.1.3　V4L2虚拟驱动 vivi

通过前面两小节的内容，我们了解了 V4L2的接口，下面我们就应该来看看， V4L2驱动是如何具体来响应这些接口的。这里我们建议读者先回头看看图 17-1。

为了更好地抓住 V4L2驱动的骨架，我们将以 Linux 2.6源码中本身就有的虚拟视频采集设备 vivi为例，其具体代码在文件 drivers/media/video/vivi.c中；而图 17-1中 V4L2驱动核心则在文件 drivers/media/video/V4L2-dev.c中。

1.分析几个重要数据结构

vivi.c包含头文件 v4l2-device.h和 v4l2-ioctl.h，其中 v4l2-device.h中包含了 v4l2-subdev.h， v4l2-subdev.h中又包含了 v4l2-common.h， v4l2-common.h中包含了 v4l2-dev.h。

在 v4l2-dev.h中定义了结构体 video_device和 v4l2_file_operations。

在 v4l2-ioctl.h中定义了结构体 v4l2_ioctl_ops。

在 v4l2-device.h中定义了结构体 v4l2_device。

1） vivi_fops是一个 v4l2_file_operations结构体。参见代码清单 17-5。

代码清单 17-5　vivi_fops

```
static const struct v4l2_file_operations vivi_fops = {
    .owner          = THIS_MODULE,
    .open           = vivi_open,
    .release        = vivi_close,
    .read           = vivi_read,
    .poll           = vivi_poll,
    .ioctl          = video_ioctl2, /* V4L2 ioctl handler */
    .mmap           = vivi_mmap,
};
```

video_ioctl2这个功能函数，其实现在源文件 drivers/media/video/V4L2-ioctl.c中，该功能函数其实会调用同文件下的 __video_do_ioctl（ struct file*file， unsigned int cmd， void*arg）内核函数，它会根据 file所指定以及 IOCTL命令不同而调用下面 vivi_ioctl_ops结构体的各功能函数，从而响应前面提到的各 IOCTL接口命令。

2） vivi_ioctl_ops是一个 v4l2_ioctl_ops结构体。参见代码清单 17-6。

代码清单 17-6　vivi_ioctl_ops

```
static const struct v4l2_ioctl_ops vivi_ioctl_ops = {
    .vidioc_querycap      = vidioc_querycap,
    .vidioc_enum_fmt_vid_cap  = vidioc_enum_fmt_vid_cap,
    .vidioc_g_fmt_vid_cap     = vidioc_g_fmt_vid_cap,
    .vidioc_try_fmt_vid_cap   = vidioc_try_fmt_vid_cap,
    .vidioc_s_fmt_vid_cap     = vidioc_s_fmt_vid_cap,
    .vidioc_reqbufs       = vidioc_reqbufs,
    .vidioc_querybuf      = vidioc_querybuf,
    .vidioc_qbuf          = vidioc_qbuf,
```

```
    .vidioc_dqbuf           = vidioc_dqbuf,
    .vidioc_s_std           = vidioc_s_std,
    .vidioc_enum_input      = vidioc_enum_input,
    .vidioc_g_input         = vidioc_g_input,
    .vidioc_s_input         = vidioc_s_input,
    .vidioc_queryctrl       = vidioc_queryctrl,
    .vidioc_g_ctrl          = vidioc_g_ctrl,
    .vidioc_s_ctrl          = vidioc_s_ctrl,
    .vidioc_streamon        = vidioc_streamon,
    .vidioc_streamoff       = vidioc_streamoff,
#ifdef CONFIG_VIDEO_V4L1_COMPAT
    .vidiocgmbuf            = vidiocgmbuf,
#endif
};
```

3） vivi_template是一个 video_device结构体，它也是视频采集设备驱动层的核心结构体。参见代码清单 17-7。

代码清单 17-7　vivi_template

```
static struct video_device vivi_template = {
    .name        = "vivi",
    .fops           = &vivi_fops,
    .ioctl_ops     = &vivi_ioctl_ops,
    .minor        = -1,
    .release    = video_device_release,

    .tvnorms              = V4L2_STD_525_60,
    .current_norm         = V4L2_STD_NTSC_M,
};
```

其中函数 vivi_xxx和 vidioc_xxx都是在 vivi.c中实现的。如果要基于某个硬件来实现 V4L2的接口，那这些函数就需要在具体的硬件驱动去实现。

4） vivi_dev，其是 vivi驱动使用的 private结构体，是对 vivi设备的一个抽象。上层对该设备进行访问时，都会以该结构体对象为该具体 vivi设备的私有数据；正是这样一个 vivi驱动得以向多个 vivi设备提供服务。参见代码清单 17-8。

代码清单 17-8　vivi_dev

```
struct vivi_dev {
    struct list_head           vivi_devlist; //内核设备双向链表
    struct v4l2_device         v4l2_dev;     //存放着作为V4L2设备的相
关信息

    /* controls */
    int                brightness; // 图像亮度
    int                contrast; // 图像对比度
    int                saturation; // 图像饱和度
    int                hue; // 图像色调
    int                volume; // 音量

    spinlock_t                 slock; // 自旋锁,实现视频数据的同步访
问
    struct mutex           mutex; // 互斥锁,实现设备控制信息的同步访
问

    /* various device info */
    struct video_device        *vfd; //这个成员是这个结构的核心,用面
向对象的话来说就是基类

    struct vivi_dmaqueue       vidq; //DMA队列

    /* Several counters */
    unsigned            ms;
    unsigned long              jiffies;

/* Controls bars movement */
    int                mv_count;

    /* Input Number */
    int                input;

    /* video capture */
    struct vivi_fmt            *fmt;     // 下面的都是vivi设备的私有
成员
    unsigned int               width, height;
    struct videobuf_queue      vb_vidq;

    unsigned long            generating;
    u8                bars[9][3];
```

```
    u8                  line[MAX_WIDTH * 4];
};
```

像这样定义的结构在 Linux C代码中很普遍，这是面向对象编程思想在 Linux的运用。实例化这个结构的对象之后，所有的操作都是基于这个结构或者这个结构派生出来的其他结构。

2.驱动初始化与设备的注册

vivi驱动初始化的主要任务就是向系统挂载相应的 vivi设备。这个流程的主线就如图 17-1的两条短虚线箭头所示。参见代码清单 17-9。

代码清单 17-9　vivi驱动初始化与设备注册

```
static int __init vivi_init(void)
{
    ...

    if (n_devs <= 0)    // n_devs记录着要创建vivi设备的数目
        n_devs = 1;

    for (i = 0; i < n_devs; i++) {
        ret = vivi_create_instance(i); // 创建若干个vivi设备实例
        if (ret) {
            /* If some instantiations succeeded, keep driver */
            if (i)
                ret = 0;
            break;
        }
    }

    ...
}
static int __init vivi_create_instance(int inst)
{
    ...

    snprintf(dev->v4l2_dev.name, sizeof(dev->v4l2_dev.name),
            ?%s-%03d?, VIVI_MODULE_NAME, inst);
    ret = v4l2_device_register(NULL, &dev->v4l2_dev);
    if (ret)
        goto free_dev;
```

```
    dev->fmt = &formats[0];     // 设定视频图像数据格式等相关信息
    dev->width = 640;
    dev->height = 480;
    dev->volume = 200;
    dev->brightness = 127;
    dev->contrast = 16;
    dev->saturation = 127;
    dev->hue = 0;

    // 初始化视频采集要用到的帧缓存,以及视频采集要用到的环形队列,并指定了该
队列的具体操作
videobuf_queue_vmalloc_init(&dev->vb_vidq, &vivi_video_qops,
            NULL, &dev->slock, V4L2_BUF_TYPE_VIDEO_CAPTURE,
            V4L2_FIELD_INTERLACED,
            sizeof(struct vivi_buffer), dev);

    /* init video dma queues */
    INIT_LIST_HEAD(&dev->vidq.active);     // 采用DMA的方式来抓取视
频数据
    init_waitqueue_head(&dev->vidq.wq);

    /* initialize locks */
    spin_lock_init(&dev->slock);
    mutex_init(&dev->mutex);

    ret = -ENOMEM;
    vfd = video_device_alloc();
    if (!vfd)
        goto unreg_dev;

    *vfd = vivi_template;
    vfd->debug = debug;
    vfd->v4l2_dev = &dev->v4l2_dev;

    ret = video_register_device(vfd, VFL_TYPE_GRABBER,
video_nr); // 向V4L2核心注册一video设备
    if (ret < 0)
        goto rel_vdev;

    video_set_drvdata(vfd, dev);

    /* Now that everything is fine, let's add it to device list
*/
    list_add_tail(&dev->vivi_devlist, &vivi_devlist);
```

```
    ...
}
```

其中所调用函数 video_register_device（）实际调用的是同文件drivers/media/video/V4L2-dev.c的内核函数——__video_register_device（）。参见代码清单 17-10。

代码清单 17-10　__video_register_device（）

```
static int __video_register_device(struct video_device *vdev,
int type, int nr, int warn_if_nr_in_use)
{
    ...

    /* v4l2_fh 支持,用以记录视频文件句柄 */
    spin_lock_init(&vdev->fh_lock);
    INIT_LIST_HEAD(&vdev->fh_list);

    /* Part 1:检查设备类型 */
    switch (type) {
    case VFL_TYPE_GRABBER:
        name_base = "video";
        break;
    ...
    default:
        printk(KERN_ERR "%s called with unknown type: %d\n",
                __func__, type);
        return -EINVAL;
    }

    vdev->vfl_type = type;
    vdev->cdev = NULL;
    if (vdev->v4l2_dev && vdev->v4l2_dev->dev)
        vdev->parent = vdev->v4l2_dev->dev;

    /* Part 2:查找没用的次设备号*/
#ifdef CONFIG_VIDEO_FIXED_MINOR_RANGES
/* 兼容老版本的4类V4L设备,次设备号前面的号码被预留
     * 而新型的新视频设备则要从128～191号段中选,而且靠前的号码将被优先选取
*/
    switch (type) {
      case VFL_TYPE_GRABBER:
        minor_offset = 0;
```

```
        minor_cnt = 64;
        break;
      ...
      default:
        minor_offset = 128;
        minor_cnt = 64;
        break;
    }
#endif

    /* Pick a device node number */
    mutex_lock(&videodev_lock);
    nr = devnode_find(vdev, nr == -1 ? 0 : nr, minor_cnt);
    ...
    mutex_unlock(&videodev_lock);

    /* Part 3:初始化为字符类设备*/
    vdev->cdev = cdev_alloc();    // 我们的V4L2设备是个字符类的设备
    ...
    vdev->cdev->owner = vdev->fops->owner;
    ret = cdev_add(vdev->cdev, MKDEV(VIDEO_MAJOR, vdev->minor),
1); // 向系统增加该vivi字符设备
    ...

    /* Part 4: register the device with sysfs */
    vdev->dev.class = &video_class;
    vdev->dev.devt = MKDEV(VIDEO_MAJOR, vdev->minor);
    ...
    dev_set_name(&vdev->dev, "%s%d", name_base, vdev->num);
    ret = device_register(&vdev->dev);
    ...
    /* Register the release callback that will be called when
the last reference to the device goes away. */
    vdev->dev.release = v4l2_device_release;

    ...

    /* Part 5: Activate this minor. The char device can now be
used. */
    set_bit(V4L2_FL_REGISTERED, &vdev->flags);
    mutex_lock(&videodev_lock);
    video_device[vdev->minor] = vdev;
    mutex_unlock(&videodev_lock);
    return 0;
}
```

至此， vivi设备驱动就完成了初始化，并向系统成功注册。上层用户就可以通过该设备来捕获想要的视频数据了。

3.数据的传输

由于本设备是一个虚拟设备，其并没有实际的视频数据产生，其实是通过程序向帧缓存填充数据来仿真的。

当上层应用通过调用 read、 poll或 IOCTL的 VIDIOC_STREAMON命令， vivi驱动将会启动线程 vivi_thread，该线程会去等待 DMA上传来的数据。

17.2　Android V4L2实践

17.1节中讲述了 LinuxV4L2子系统普适性的内容。下面我们就通过 Android中具体 V4L2设备来看看， Android基于 V4L2的应用与实践。

以下示例基于 Marvell PXA910的 CPU、 OV5642 Camera模组。

17.2.1　硬件基础

在这个例子中， 500M分辨率的 OV5642 Camera模组，通过 PXA910的 CMOS Camera接口控制器（ CMOS Camera Interface Controller，CCIC）向上层应用提供预览、拍照、录像要用的视频采样；而且 PXA910通过 I2C实现对 OV5642模组工作模式选择、闪光灯控制、工作状态查询等。其硬件框架如图 17-4所示。

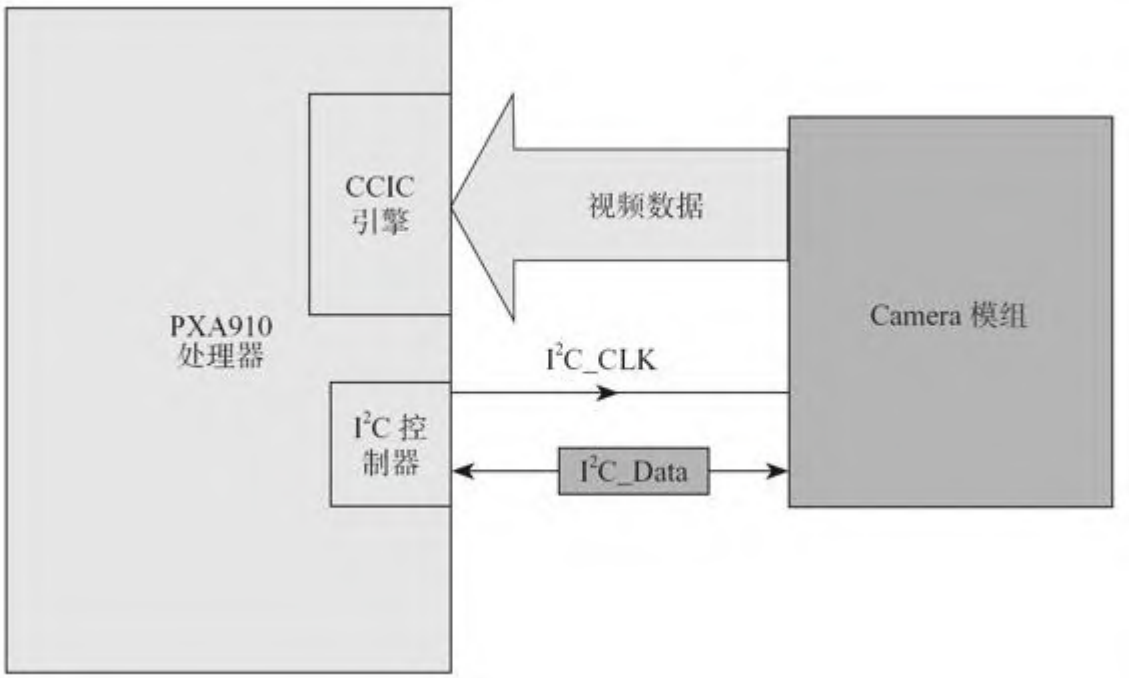


图17-4　PXA910与OV5642模组接口框图

图 17-5是 PXA910CCIC的功能框图。该图左侧是与我们 Camera模组的接口，对 PXA910 CCIC来讲是输入。 PXA910提供了两种接口：一个是 MIPI CSI-2；另一个是并行的接口。在我们的设计中选用的是并行接

口 CCIC-0。图的右侧是与 PXA910 CPU的接口，该接口实现了将 CCIC采样到的视频数据传送给 PXA910 CPU处理；另外还实现了 PXA910 CPU对 CCIC模块的控制以及 CCIC模块工作状态的上报。

针对 CCIC-0并行接口，其通过图 17-6的时序实现了 Camera数据的采样。

OV5642模组是基于 OmniVision公司的 OV5642 ISP处理芯片，实现500M像素的图像视频采样；另外 OV5642模组上还集成了闪光灯控制，可以为 Camera闪光灯的控制节省 GPIO等相关资源。在我们选用的OV5642模组中，还集成了一个 Motor驱动芯片，用于实现该模组拍照时的自动调焦功能。

由于在我们的硬件系统有 CCIC与 Camera模组两个相对独立的子系统，因此在 Android这部分的设计中，在遵循 V4L2架构的同时，我们将其分成两个模块来实现 Android系统的视频采集功能。

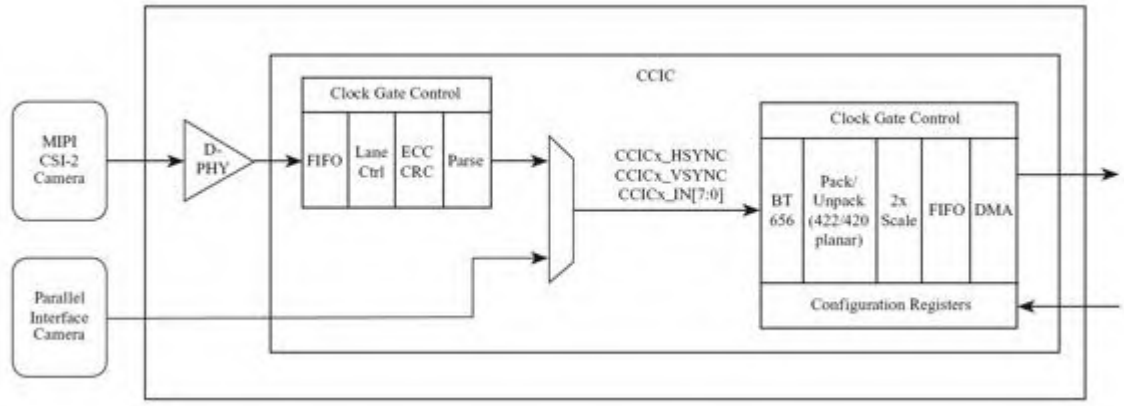


图17-5　CCIC功能框图

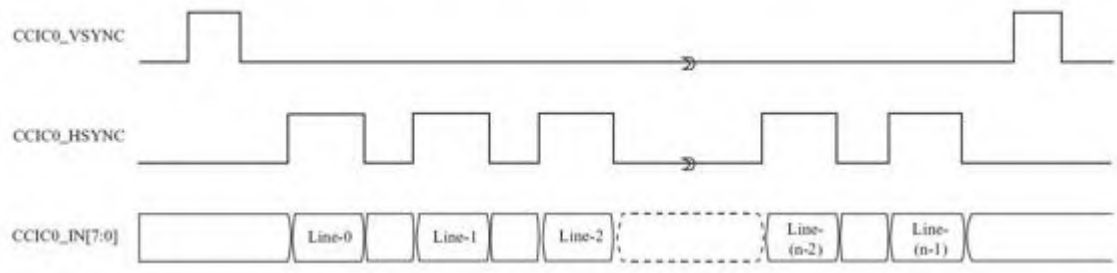


图17-6　CCIC并行采样数据图

17.2.2 CPU侧 CCIC驱动模块

该模块我们主要实现在文件 pxa910_camera.c中实现与 V4L2子系统核心的接口，同时也实现针对 PXA910 CPU CCIC的驱动。

首先， CCIC作为挂载在 PXA910 CPU总线上的一个模块，是 Platform虚拟总线上的一个设备。因此，在我们的驱动里应该将该驱动注册到Platform虚拟总线。参见代码清单 17-11。

代码清单 17-11 pxa910 CCIC初始化

```
static int __devinit pxa910_camera_init(void)
{
    spin_lock_init(&reg_lock);
    wake_lock_init(&idle_lock, WAKE_LOCK_IDLE, ?
pxa910_camera_idle?);

    ...

    return platform_driver_register(&pxa910_camera_driver);
}
```

其中， platform_driver_register（）函数会调用本模块下的pxa910_camera_probe（），而 pxa910_camera_probe（）又会去调用video_register_device（），以将本驱动所对应的视频设备注册到V4L2子系统中。参见代码清单 17-12。

代码清单 17-12 pxa910_camera_probe（）

```
static int pxa910_camera_probe(struct platform_device *pdev)
{
    ...
    struct ccic_camera *cam;
    /*
     *下面开始初始化大的数据结构struct ccic_camera实例：cam
     */
    ret = -ENOMEM;
    rst_clk = clk_get(&pdev->dev, "CCICRSTCLK");
```

```
    ...
    pxa168_ccic_gate_clk = clk_get(&pdev->dev, "CCICGATECLK");
    ...

    cam = kzalloc(sizeof(struct ccic_camera), GFP_KERNEL);
    ...
    cam->lcd_clk = clk_get(NULL, "LCDCLK");
    ...
    clk_enable(cam->lcd_clk);

    platform_set_drvdata(pdev, cam);
    mutex_init(&cam->s_mutex);
    /*在probe()函数中不需要锁定该驱动的mutex,因为我们是在late_init阶段
才真正启动该控制所控制的sensor*/
    spin_lock_init(&cam->dev_lock);
    cam->state = S_NOTREADY;
    init_waitqueue_head(&cam->iowait);
    cam->pdev = pdev;
    INIT_LIST_HEAD(&cam->dev_list);
    INIT_LIST_HEAD(&cam->sb_avail);
    INIT_LIST_HEAD(&cam->sb_full);
    tasklet_init(&cam->s_tasklet, ccic_frame_tasklet, (unsigned
long) cam);

    cam->irq = platform_get_irq(pdev, 0);
    ...

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    ...
    cam->regs = ioremap(res->start, SZ_4K);
    ...
    ret = request_irq(cam->irq, ccic_irq, IRQF_SHARED, "pxa910-
camera", cam);
    ...
    /*
     *初始化CCIC控制器并给它上电
     *该控制器会保持当前状态不变,直至sensor驱动被启动运行
     */
    ccic_ctlr_init(cam);
    ccic_ctlr_power_up(cam);
    ...
    /*
     * 配置v4l2
     */
    cam->v4ldev = ccic_v4l_template;
    cam->v4ldev.debug = 0;
    memset(&cam->v4ldev.dev, 0, sizeof(cam->v4ldev.dev));
```

```
    cam->v4ldev.index = pdev->id;
    printk(KERN_NOTICE "camera_probe id %d\n", cam-
>v4ldev.index);

    cam->v4ldev.parent = &pdev->dev;
    ret = video_register_device(&cam->v4ldev, VFL_TYPE_GRABBER,
-1);
    ...
    cam->ccic_early_suspend.level =
EARLY_SUSPEND_LEVEL_STOP_DRAWING,
    cam->ccic_early_suspend.suspend = ccic_sleep_early_suspend;
    cam->ccic_early_suspend.resume = ccic_normal_late_resume;
    register_early_suspend(&cam->ccic_early_suspend);

    ccic_add_dev(cam);
/*    ccic_ctlr_power_down(cam); //将CCIC的供电断掉,从节省功耗方面考
虑,可以打开该语句*/
#ifdef CONFIG_PM
    camera_pin_power_opt();
#endif
    return 0;

out_freeirq:
    ccic_ctlr_power_down(cam);
    free_irq(cam->irq, cam);
out_iounmap:
    iounmap(cam->regs);
out_free:
    kfree(cam);
out:
    return ret;
}
```

从代码清单 17-12可以看到，在该函数中我们不仅要完成视频设备的注册，还需要完成 CCIC所需要的以下资源配置：时钟、中断、管脚、I/O内存映射等，还会有对该控制器的供电控制。

17.2.3　OV5642模组驱动模块

要想真正从 CCIC提取到图像数据，仅有 CCIC这个控制器和它的驱动是不行的，还需要光感应器和相应的设备驱动。我们这里用的光感应器就是 OV5642。从图 17-7可以看到， OV5642通过 CCIC-0并行总线，向 PX910的 CCIC提供采样到的视频图像数据；同时，还可以看到

该模组通过 I2C_SCL与 I2C_SDA，挂载到了 CPU的 I2C总线上。为了让大家更清楚后面我们针对 PXA910 OV5642V4L2驱动的理解，下面给出实际的 OV5642的接口图（见图 17-7）。

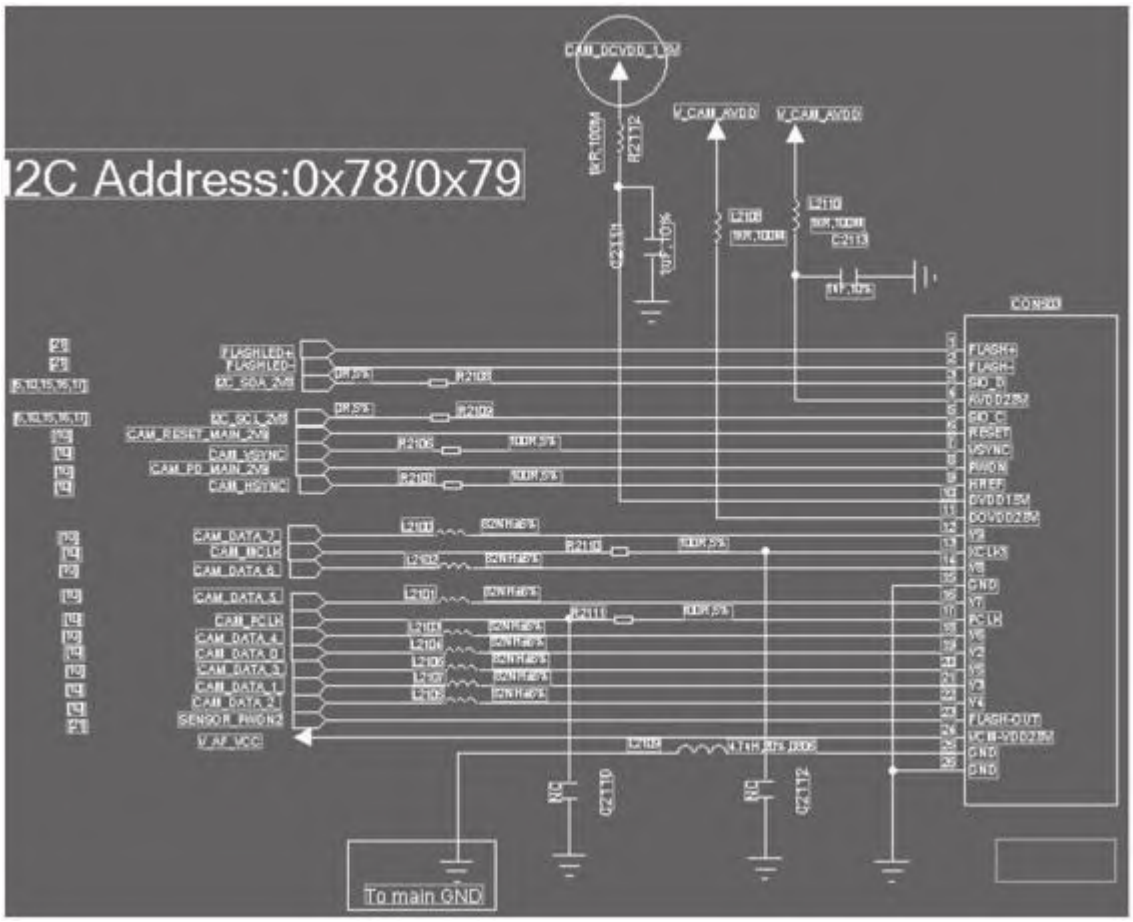


图17-7　与OV5642的相关的硬件原理图

事实上，从驱动工作机制来看， OV5642模组更是一个挂载在 I2C总线上的字符类设备。该字符类设备，通过 I2C总线，读 /写 OV5642模组的控制寄存器，控制着 OV5642的采样格式、调焦马达工作、闪光灯的开关等。为了实现该模组的工作，我们应该了解其工作机制和相应寄存器的定义，这里我们就不详述了，感兴趣的读者可以从 OV的官方网站下载相关的资料。该 OV5642的驱动代码骨架如代码清单 17-13。

代码清单 17-13　 OV5642驱动骨架代码

```
static int __init ov5642_mod_init(void)
{
    int ret = 0;
    ret = i2c_add_driver(&ov5642_driver);
    return ret;
}
static void __exit ov5642_mod_exit(void)
{
    i2c_del_driver(&ov5642_driver);
}
static struct i2c_driver ov5642_driver = {
    .driver = {
           .name = "ov5642",
           },
    .probe = ov5642_probe,
    .remove = ov5642_remove,
    .id_table = ov5642_idtable,
};
static int ov5642_probe(struct i2c_client *client, const struct
i2c_device_id *did)
{
    struct ov5642 *ov5642;
    struct soc_camera_device *icd = client->dev.platform_data;
    struct i2c_adapter *adapter = to_i2c_adapter(client-
>dev.parent);
    struct soc_camera_link *icl;
    int ret;

    g_i2c_client = client;
    ...
    icl = to_soc_camera_link(icd);
    ...
    if (!i2c_check_functionality(adapter,
I2C_FUNC_SMBUS_WORD_DATA)) {
        ...
    }
    ov5642 = kzalloc(sizeof(struct ov5642), GFP_KERNEL);
    ...
    v4l2_i2c_subdev_init(&ov5642->subdev, client,
&ov5642_subdev_ops);
    icd->ops = &ov5642_ops;
    ov5642->rect.left = 0;
    ov5642->rect.top = 0;
    ov5642->rect.width = 640;
    ov5642->rect.height = 480;
    ov5642->pixfmt = V4L2_PIX_FMT_YUV422P;
    ret = ov5642_video_probe(icd, client);
```

```
    ...
}
static int ov5642_remove(struct i2c_client *client)
{
    ...
    kfree(ov5642);
    return 0;
}
static const struct i2c_device_id ov5642_idtable[] = {
    {?ov5642?, 0},
    {}
};
static struct v4l2_subdev_core_ops ov5642_subdev_core_ops = {
    .g_chip_ident = ov5642_g_chip_ident,
    .load_fw = ov5642_load_fw,
    .s_ctrl    = ov5642_s_ctrl,
    ...
};
static struct v4l2_subdev_video_ops ov5642_subdev_video_ops = {
    .s_stream = ov5642_s_stream,
    .g_mbus_fmt = ov5642_g_fmt,
    .s_mbus_fmt = ov5642_s_fmt,
    .try_mbus_fmt = ov5642_try_fmt,
    .enum_mbus_fsizes = ov5642_enum_fsizes,
    .enum_mbus_fmt = ov5642_enum_fmt,
};
static struct v4l2_subdev_ops ov5642_subdev_ops = {
    .core = &ov5642_subdev_core_ops,
    .video = &ov5642_subdev_video_ops,
};
static struct soc_camera_ops ov5642_ops = {
    .query_bus_param = ov5642_query_bus_param,
    .set_bus_param = ov5642_set_bus_param,
    .controls            = ov5642_controls,
    .num_controls        = ARRAY_SIZE(ov5642_controls),
};
```

各操作函数的实现这里就不再详述了。我相信，基于前面的学习理解这个驱动不难。

17.3 Android系统对 V4L2的使用

关于 Android系统中对 V4L2子系统的使用，我们还是以 Camera为例。在 Android机器中 Camera应用实现了拍照、录像与预览三大照相机的基本功能。这些功能的实现由前面讲的 CCIC驱动模块落实。

同样，按第 14章所述，为了让 Android上层利用到 Camera驱动、利用 Camera模组采集想要的视频图像数据，我们也应在 Android的 Framework实现相应的 HAL与 Service，如图 17-8所示。

图中 Libcamera.so就是 Camera HAL的库，里面存放有调用 V4L2驱动的接口，以及 Jpeg编码程序。该 HAL实现的接口要符合 CameraHardwareInterface规定，一般由各 CPU厂商提供，实现对 Camera硬件的操作。

而该库又由上层的服务 libcameraservice.so，通过 frameworks/base/core/jni/android_hardware_Camera.cpp实现的 JNI来调用。 Camera应用程序如图 17-8所示，正是通过第 18章要讲的 Binder IPC来与 Camera Sevice交互，进而实现对 Camera的访问。

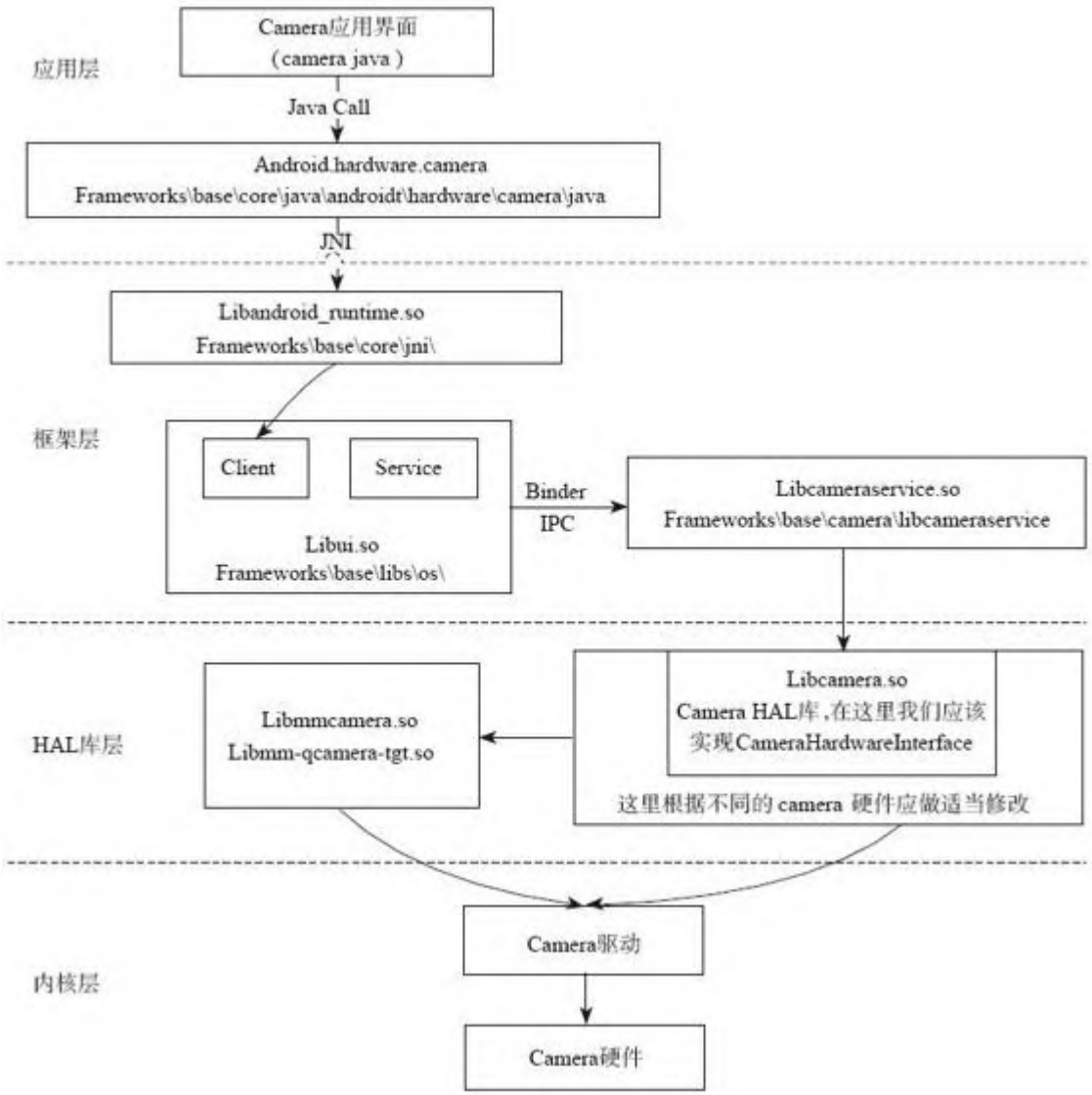


图17-8　Android Camera系统架构图

第18章　Binder IPC通信子系统

前面讲的都是一般Linux都有的驱动模块，本章我们介绍一个Android专有的驱动子系统——Binder IPC通信子系统。其中我们在前面，多次讲到各子系统对Binder IPC通信子系统的运用。

在本章我们会介绍该子系统的实现，但讲的更多的是它的工作机理，因为作为读者新建或修改Binder驱动的机会不多，主要都是在使用它。

18.1 Binder驱动概述

前面讲过，在 linux中进程间的通信机制有很多种，如管道（ pipe）、消息队列（ message queue）、信号（ signal）、共享内存（ share memory）、套接字（ socket）等方式。但是，在 Android终端上应用软件的通信几乎看不到这些 IPC通信方式，取而代之的是Binder。 Android同时为 Java环境和 C/C++环境提供了 Binder机制。

Binder是一套轻量型的 IPC机制，它比 linux一般的通信方式更加简洁，消耗的内存资源更小。 Binder主要提供以下功能：

1）用驱动程序来推进进程间的通信。

2）通过共享内存来提高性能。

3）为进程请求分配每个进程的线程池。

4）针对系统中的对象引入了引用计数和跨进程的对象引用映射。

5）进程间同步调用。

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

18.2 Binder通信模型

Binder通信机制在用户空间为每个进程维护着一个可用的线程池，线程池用于处理到来的 IPC以及执行进程的本地消息。 Binder通信是同步的。同时， Binder通信机制是基于 Android专有 Linux驱动 Binder来实现的，从而导致 Android系统本身的运行都将依赖 Binder驱动。

Binder通信是 C/S模式，也就是基于服务器（ Service）与客户端（ Client）的，所以每个需要 Binder通信的进程都必须创建一个 Binder接口。 Android系统通过名为 Service Manager的守护进程管理着系统中的各个服务，它负责监听是否有其他程序向其发送请求，如果有请求就响应，如果没有则继续监听等待。每个服务都要在 Service Manager中注册，而请求服务的客户端则向 Service Manager请求服务。在 Android虚拟机启动之前，系统会先启动 Service Manager进程， Service Manager就会打开 Binder设备，并通知内核中 Binder驱动程序，接着该进程将扮演系统服务的管理者，其会进入一个循环，等待处理来自其他进程的数据。因此， Binder IPC子系统的实现牵涉到 Binder驱动等部分。

- Binder驱动为上层应用程序和用户操作提供各种操作接口。

- Service Manager主要负责管理 Android系统中所有服务，当客户端要与服务器端进行通信时，首先就会通过 Service Manager来查询和取得所需要交互的服务。当然，每个服务器也都需要向 Service Manager注册自己提供的服务，以便能够供客户端进行查询和获取。

- Server这里的服务即上面所说的服务器端，通常也是 Android的系统服务，通过 Service Manager可以查询和获取某个 Server。

- Client这里的客户端一般是指 Android系统上面的应用程序。它可以请求 Server中的服务，比如 Activity就可以是这样的一个 Client。

图 18-1显示了 Binder IPC通信的原理。

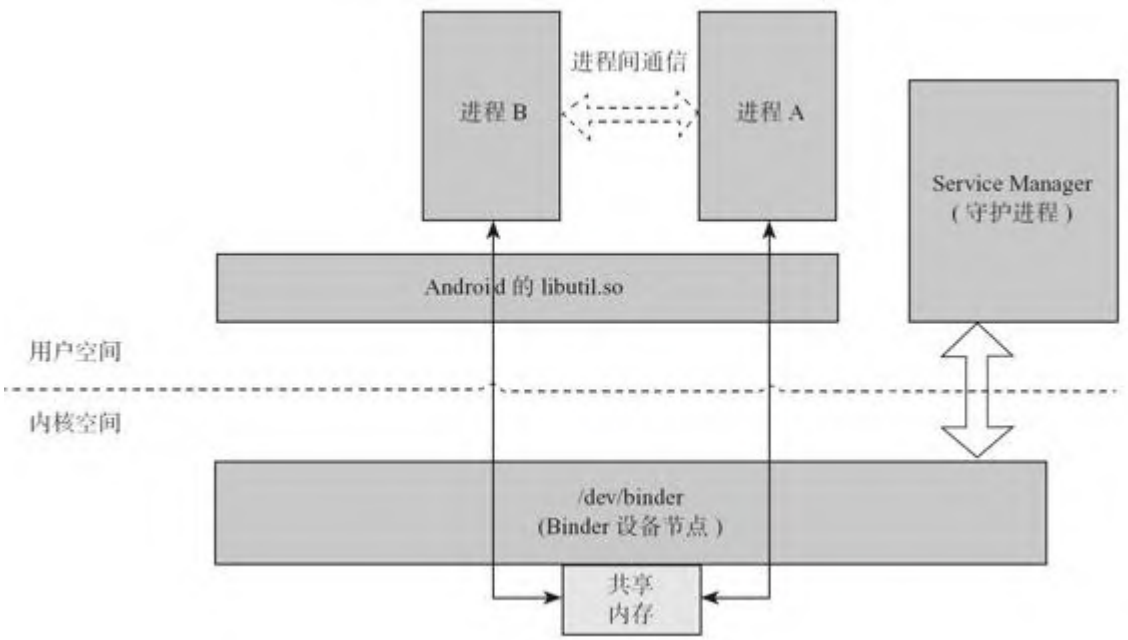


图18-1　Binder通信原理图

基于 Client-Server（ C/S）的交互模式，可以跨计算机平台，广泛应用于 PC访问互联网和网上数据库，也可以用于嵌入式手持设备内部进程间的交互。智能手机平台特别是 Android系统，为了向应用开发者提供丰富多样的功能，这种通信方式更是无处不在，诸如媒体播放、视音频捕获等，都由不同的 Server负责管理；应用程序只需为 Client与 Server建立连接便可以使用这些服务，结果程序员花很少的时间和精力就能开发出令人目眩的功能。但 Client-Server方式的广泛采用，需要进程间通信（ IPC）技术支持。标准 linux支持的 IPC包括传统的管道、 System V IPC，即消息队列 /共享内存 /信号量，以及 socket等。其中只有 socket支持 Client-Server的通信方式。当然也可以在其他底层通信机制上新开发一套协议来实现 Client-Server通信，但这样增加了系统的复杂性，对于处于移动互联环境恶劣、内存资源稀缺的手持终端设备，这种改造得不偿失。 Android Binder IPC通信子系统就应这种需求而产生了。

假设一个进程作为 Server提供诸如视频 /音频解码、视频捕获、地址本查询、网络连接等服务；多个进程作为 Client向 Server发起服务请求，获得所需要的服务。要想实现 Client-Server通信必须实现以下两点功能：一是 Server必须有确定的访问接入点或者地址来接收 Client的请求，并且 Client可以通过某种途径获知 Server的地址；

二是制定 Command-Reply协议来传输数据。例如在网络通信中 Server的访问接入点就是 Server主机的 IP地址 +端口号，传输协议为 TCP协议。而 Android的 Binder很好地实现了这两个功能。对 Server而言， Binder可以看成 Server提供的实现某个特定服务的访问接入点， Client通过这个“地址”向 Server发送请求来使用该服务；对Client而言， Binder可以看成是通向 Server的管道入口，其与某个Server通信前必须先建立这个管道并获得管道入口。

在基于 Binder驱动的 IPC通信机制中，我们可以看到面向对象思想的光辉： Binder是一个实体位于 Server中的对象，该对象提供了一套方法用以响应对服务的请求，就像类的成员函数。遍布于 Client中的入口可以看成指向这个 Binder对象的“指针”，一旦获得了这个“指针”，就可以调用该对象的方法访问 Server。在 Client看来，通过Binder“指针”调用其提供的方法和通过指针调用其他任何本地对象的方法并无区别，尽管前者的实体位于不同进程的 Server中，而不像本地对象处在 Client所在进程的本地内存中。“指针”是 C++的术语，其实这里更恰当的讲法应是“引用”，即 Client通过 Binder的引用访问 Server。而软件领域另一个术语“句柄”也可以用来表述Binder在 Client中的存在方式。从通信的角度看， Client中的Binder也可以看作是 Server Binder的“代理”，在本地代表“远端” Server为 Client提供服务。在后面关于 Binder的描述中，会更多地使用“引用”或“句柄”这两个广泛使用的术语。

至此，我们可以清楚看到 Binder IPC与其他 Linux IPC不同，其使用了面向对象的思想，来描述作为访问接入点的 Binder，及其在Client中的入口。面向对象思想的引入将进程间通信转化为通过对某个 Binder对象的引用调用该对象的方法，而其独特之处在于 Binder对象是一个可以跨进程引用的对象，它的实体位于一个进程中，而它的引用却遍布于系统的各个进程之中。同时，这个引用和 Java里引用一样既可以是强类型，也可以是弱类型，而且可以从一个进程传给其他进程，让大家都能访问同一 Server，就像将一个对象或引用赋值给另一个引用一样，这使程序的逻辑更加清晰，程序的开发与升级更加简单。 Binder模糊了进程边界，淡化了进程间通信过程，因此整个系统仿佛运行于同一个面向对象的程序之中。像 Binder-英文里的原意“胶水”，各个 Binder对象被方便地“粘合”在一起（对象引用），很方便地就开发出丰富多彩的应用程序。可以讲这就是Android蓬勃兴起的主要原因之一。

当然面向对象只是针对应用程序而言，对于 Binder驱动而言，与其他内核模块一样，其使用 C语言实现，除了运用了面向对象的思想，并没有类和对象定义或实现。总之， Binder驱动为面向对象的进程间通信提供底层支撑。

18.3 Binder驱动

Binder驱动程序的部分在以下文件夹中：

- kernel/include/linux/binder.h

- kernel/drivers/Android/binder.c

Binder驱动程序是一个 miscdevice的驱动，主设备号为 10，此设备号使用动态获得（ MISC_DYNAMIC_MINOR），其设备的节点为/dev/binder。

在其驱动的实现过程中，主要通过 binder_ioctl函数与用户空间的进程交换数据。 BINDER_WRITE_READ用来读写数据；而且在交互的数据包中有一个 cmd域用于区分不同的请求。 binder_thread_write（）函数用于发送请求或返回结果，而 binder_thread_read（）函数则用于读取结果。在 binder_thread_write（）函数中调用binder_transaction（）函数来转发请求并返回结果。当收到请求时， binder_transaction（）函数会通过对象的 handle，找到对象所在的进程；如果 handle为空，就认为对象是 context_mgr，把请求发给 context_mgr所在的进程。请求中将所有的 Binder对象全部放到一个 RB树中，最后把请求放到目标进程的队列中等待目标进程读取。数据的解析工作放在 binder_parse（）中实现。关于如何生成context_mgr，内核中提供了 BINDER_SET_CONTEXT_MGR命令来完成此项功能。

有了前面的学习， Binder驱动的理解不难。有感兴趣的读者可进一步查阅 ASOP相关源码。

18.3.1 Binder相关的结构体

这些结构体的具体定义，在 AOSP的源码中都可以找到，这里就不全部列出，而是概括它们主要的内容与作用，为后续的理解做铺垫。

- binder_work

该结构体是存放进程工作项链表，同时记录着各工作项的工作状态。

• flat_binder_object

我们把进程之间传递的数据称为 Binder对象（ Binder Object），它在对应源码中使用 flat_binder_object结构体（位于 binder.h文件中）来表示。

• binder_transaction_data

用于 Binder驱动的实现，以及完成进程间数据的交互。

• binder_write_read

该结构体用来存入 BINDER_WRITE_READ IOCTL命令读写的数据。

• binder_proc

binder_proc结构体用于保存调用 Binder的各个进程或线程的信息，如线程 ID、进程 ID、 Binder状态信息等。

• binder_node

该结构体表示一个 Binder节点。

• binder_thread

binder_thread结构体用于存储每一个单独线程的信息。

• binder_transaction

主要用来中转请求和返回结果，保存接收和发送的进程信息。

18.3.2 Android Binder子系统的架构设计

如图 18-2所示， Binder驱动在前面已经介绍了，它主要负责组织 Binder的服务节点，调用 Binder相关的处理线程，完成实际的 Binder传输等，它位于 Binder架构的最底层（即 Linux内核层）。 Binder Adapter层是对 Binder驱动的封装，主要用于操作 Binder驱动，即应用程序不必直接接触 Binder驱动程序，实现包括 IPCThreadState.cpp和 ProcessState.cpp以及 Parcel.cpp中的部分

内容。 Binder核心库是 Binder框架的核心实现，主要包括 IBinder、 Binder（服务器端服务管理器）和 BpBinder（客户端服务代理管理器）。而最上面两层的 Binder框架和具体的客户端 /服务端，都分别有 Java和 C++两种实现方案，主要供应用程序使用，如摄像头和多媒体等，它们通过 Binder框架调用 Binder核心库来实现。

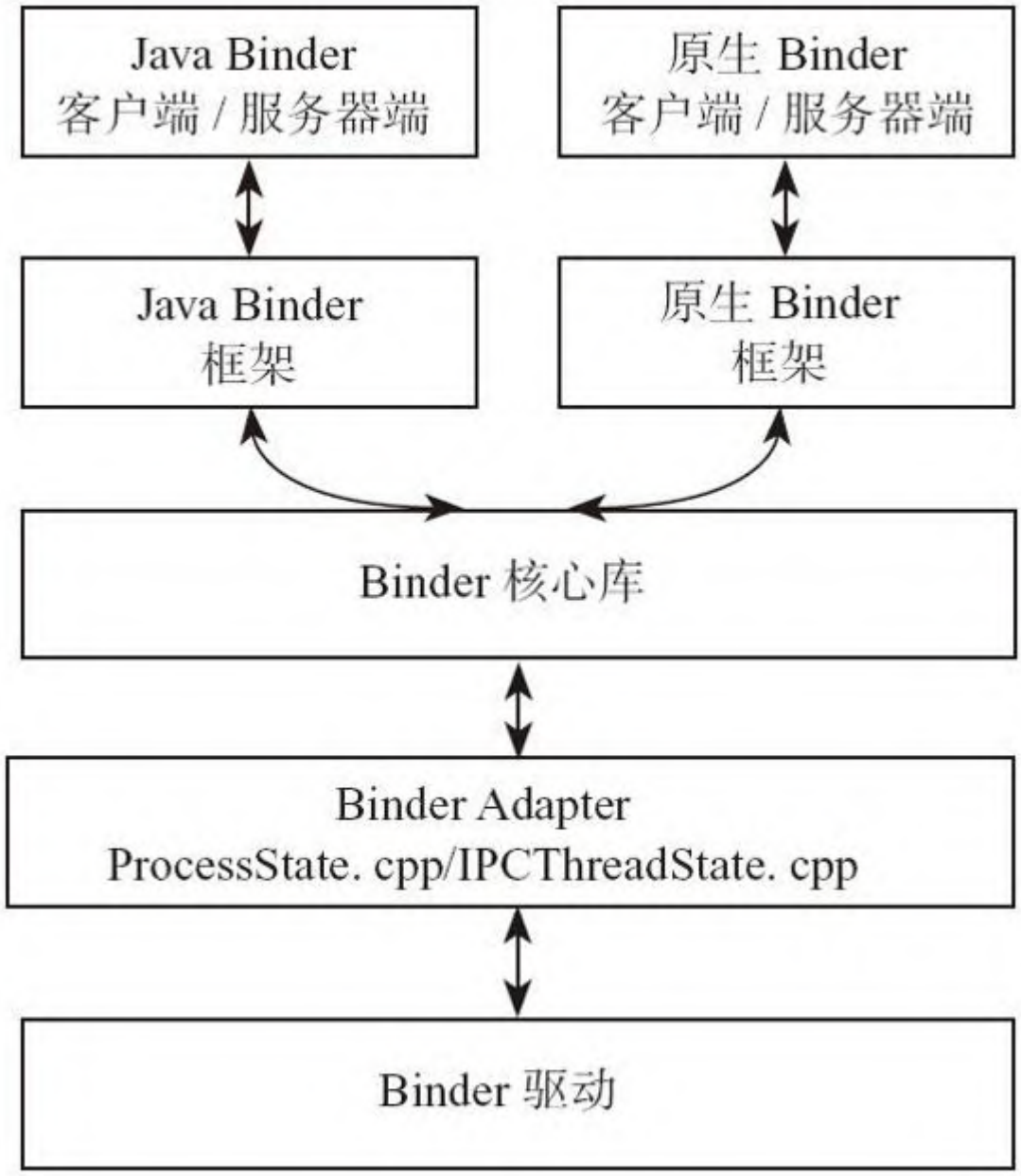


图18-2　Binder子系统实现架构

18.4 Binder的工作流程

Binder的工作流程如图 18-3所示。简要地讲，它有以下 5个主要步骤：

1）客户端首先获得服务器端服务的代理对象。所谓的代理对象实际上就是在客户端建立一个服务器端的“引用”，该代理对象具有服务器端服务的功能接口，使客户端像访问本地方法一样访问服务器端提供的服务。

2）客户端通过调用服务器代理对象向服务器端发送请求。

3）通过 Binder驱动代理对象将用户请求发送到服务器进程。

4）服务器进程处理用户请求，并通过 Binder驱动返回处理结果给客户端的服务器代理对象。

5）客户端收到服务器端的返回结果。

Binder其实是一种架构，这种架构提供了服务器端服务接口、 Binder驱动、客户端服务代理接口三个模块，以及一个守护进程 Service Manager。

首先来看服务器端。一个 Binder服务实际上就是一个 Binder类的对象，该对象一旦创建，内部就启动一个隐藏线程。该线程接下来会接收 Binder驱动发送的消息，收到消息后会执行 Binder对象中的 onTransact（）函数，并按照该函数的参数执行不同的服务代码。其中，要实现一个 Binder服务就必须重载 onTransact（）方法，以响应不同服务的不同功能代码。

可以想象，重载 onTransact（）函数的主要内容是把 onTransact（）函数的参数转换为服务函数的参数，而 onTransact（）函数的参数来源，是客户端调用 transact（）函数时输入的。 transact（）有固定格式的输入，以及固定格式的输出，以满足 Client与 Server交互的需求。

下面再看 Binder驱动。任意一个服务器端 Binder对象被创建时，同时会在 Binder驱动中创建一个 mRemote对象。客户端要访问远程服务时都是通过 mRemote对象。

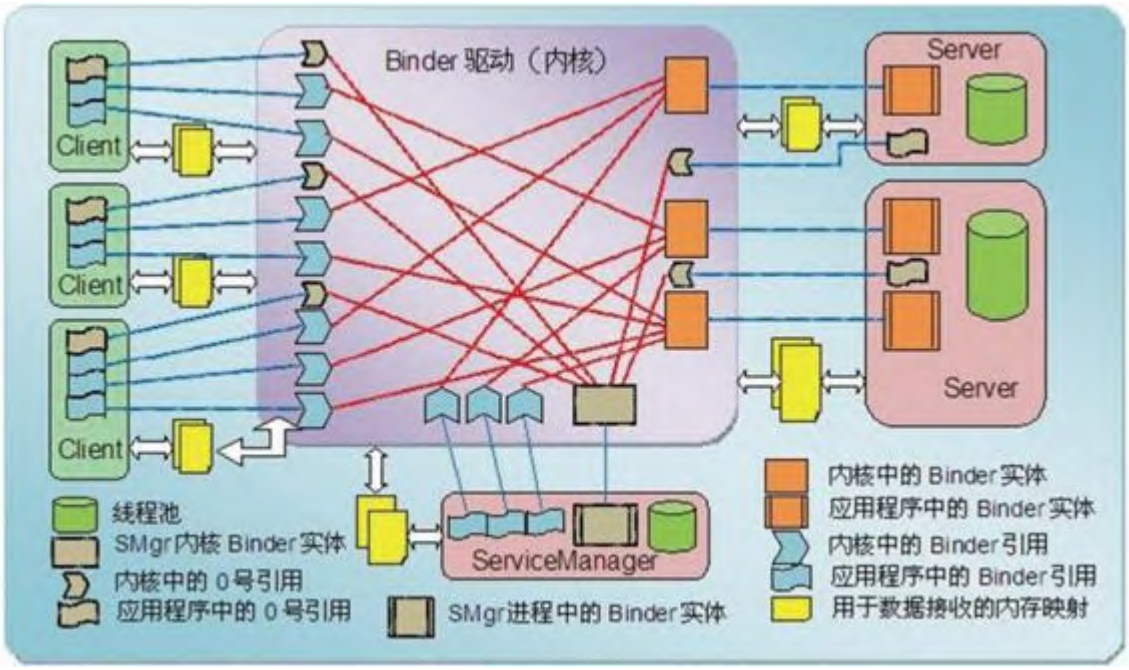


图18-3　Binder工作流程

最后来看应用程序客户端。客户端要想访问远程服务，就要获取远程服务在 Binder对象中对应的 mRemote引用。获得该 mRemote对象后，就可以调用其 transact（）方法；而在 Binder驱动中， mRemote对象也重载了 transact（）方法，重载的工作主要包括以下几项。

• 以线程间消息通信的模式，向服务器端发送客户端传递过来的参数。

• 挂起当前客户端线程，并等待服务器端线程执行完指定服务函数后发的通知（notify）。

• 接收到服务器端线程的通知，然后继续执行客户端线程，并返回到客户端代码区。

从这里可以看出，对应用程序开发员来讲，客户端似乎是直接调用远程服务对应的 Binder，而事实上基底层则是通过 Binder驱动进行了中转，而完成该中转的关键是两个 Binder对象：一个是服务器端的

Binder对象，另一个则是 Binder驱动中的 Binder对象，所不同的是Binder驱动中的对象不会再额外产生一个线程。

第19章　USB子系统

在Android智能手机中都会集成有USB，而在功能强大的机器中还会使能USB Host或OTG功能，使得Android手机不仅可以作为USB客户端与PC主机相连，还可以直接与打印机等USB设备相连，极大地丰富了Android智能手机的功能。

本章我们就准备讲讲关于USB这个有趣且复杂的子系统在Android系统中的实现。经过前面的学习，现在也是我们了解Android中更复杂子系统的时候了。

19.1　USB协议基础知识

USB本身是一个复杂的而有机的串行总线协议标准，要完整地把它讲清楚，需要专门的书籍来阐述。关于 USB协议之类的书比较多，建议对USB感兴趣的读者不妨找一些读读，比如《 USB.0原理与工程开发》。事实上，如果对 USB端点、设备描述符、接口、复合设备等概念都不清楚的话，那是无法理解 USB设备驱动与 Android USB子系统的。为了方便读者更好地理解下面的内容，我们先了解 USB的基础协议知识。

19.1.1　USB物理连接

USB是当今最流行的串行总线协议，其采用如图 19-1所示的拓扑连接。

基于该拓扑结构，我们应该关注以下特性和定义：

- 该拓扑基于主（Host控制器）/从（Device设备）体系架构。
- 该树形拓扑有效地避免了环形连接。
- 一条USB总线上有且只有一个USB Host，以及一个RootHub。
- USB设备分为两类：集线器类设备（Hub）和功能类设备（Functions），Hub通过端口（Port）连接更多的USB设备，Functions即USB外接设备，如USB鼠标等。
- 该拓扑的层次最多7层，最底层上不能有Hub设备，只能有Functions。
- 复合设备（Compound Device）：一个Hub上接多个设备组成一个小设备。
- 多功能设备（Composite Device）：一个USB外接设备具有多个功能。

现在常用的 USB的接口有很多种，如图 19-2所示。

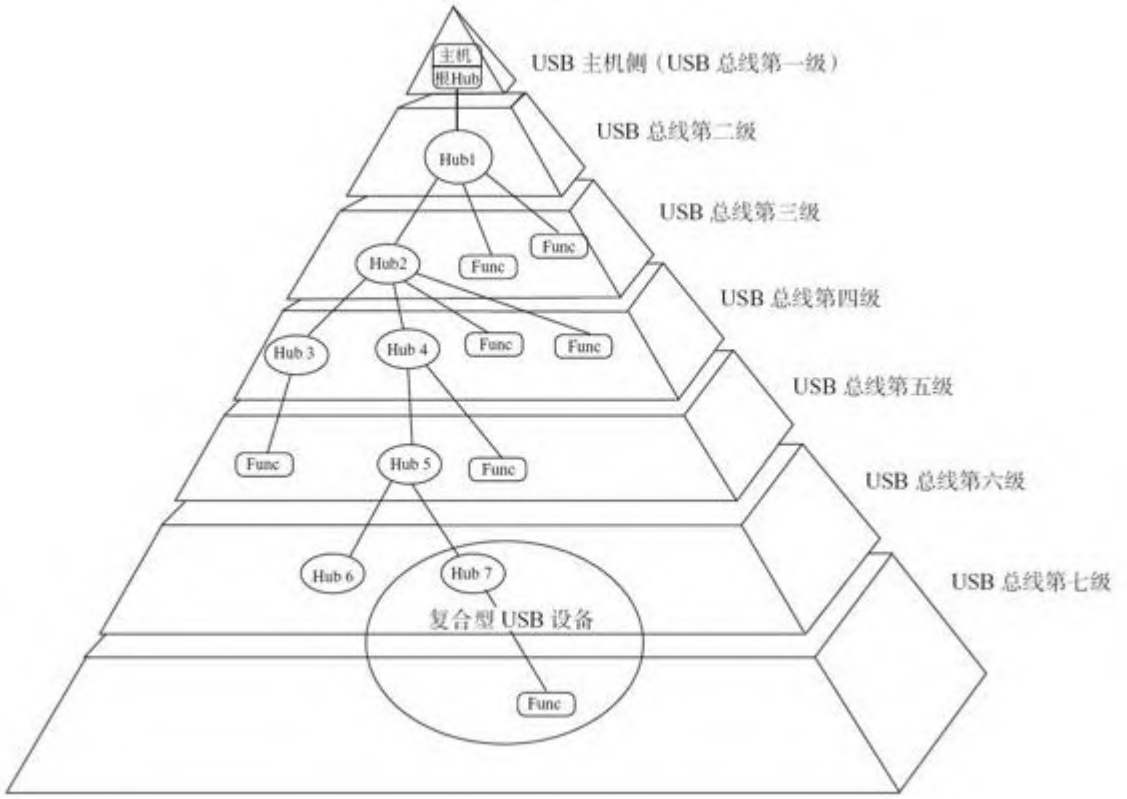


图19-1　USB总线拓扑结构

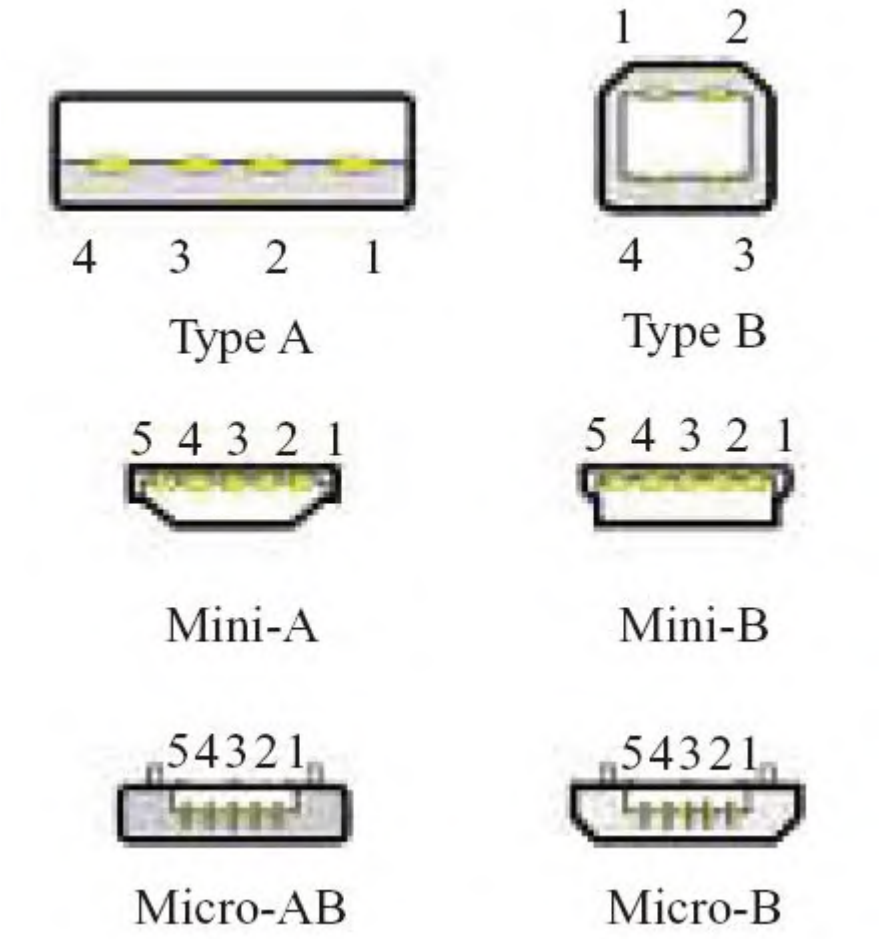


图19-2　USB连接头

其中，这里有几个概念请读者注意区分：

- 连接件connector：指设备上的那个连接口。
- 插头plug：指USB线缆两头的插口。
- Mini-AB/Micro-AB：指的支持A和B两类插头的连接件。

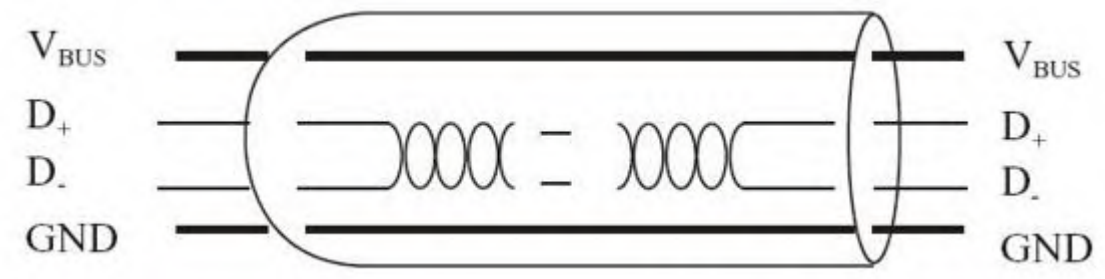


图19-3　USB电缆

USB电缆主体是 4根线，如图 19-3所示。

其中：

• VBUS：+5V电源供电。

• D+/D-：用于数据传输的双绞线。

而 USB总线上所连设备所支持的传输速率是自动识别的：

• 低速low-speed：当USB Host端检测到D-为高电平，D+为低电平，其会以10～100Kb/s的速率与相应的USB设备交互，该USB设备主要有鼠标和键盘等。

• 全速full-speed：当USB Host端检测到D-为低电平，D+为高电平，其会以500Kb/s～10Mb/s的速率与相应的USB设备交互，该USB设备主要有音频和麦克等。

• 高速high-speed：当USB Host端检测到D-为低电平，D+为低电平，其会以25～400Mb/s的速率（如果支持USB3.0的话，其速度会更快）与相应的USB设备交互，该USB设备主要用于存储和视频等。

随着 USB OTG的引入， USB电缆增加了一根 ID线，变成了 5根线。细心的读者可以从图 19-2中印证这一点。当支持 USB OTG的设备，检测到 ID脚被拉成低电平，就知道对端是 USB Host，它就会扮演 USB Client的角色；而当检测到该 ID线被悬空（为高电平），就会与对端协商，可以获得 USB Host的角色，控制与对端的通信交互。

19.1.2　USB通信协议

USB的启动与其他可热插拔设备相比，最大的不同是 Hub， USB Host通过 Hub状态的变化来判断 USB外接设备的有无。当发现有 USB外设连接后， USB Host就会给 USB外设上电，而 USB Host就会通过端点0与处于默认态的 USB外设寻址并完成配置，配置好后，就可以按照USB协议正常交互了。 USB设备启动流程如图 19-4所示。

USB外设插入和拔出整个实现过程，也就是常说的 USB总线枚举。在该枚举过程中，会用到 4类 USB描述符：设备描述符、配置描述符、接口描述符和端点描述符。 USB协议正是基于该 4类描述符识别与配置USB设备。更是基于接口和端点，实现了 USB Host与 USB Device通信的实际管道，并在该管道上传输所交互的数据。如图 19-5所示。

注：端点与端口是不同的概念。端口是 USB Hub连接其他 USB设备的物理接口。而端点是 USB协议通信的最小逻辑主体。而端点（ endpoint）又分 0端点和非 0端点。其中 0端点前面讲过，其作为默认的控制管道用于初始化和操控 USB逻辑设备。

USB数据流传输分为 4类：控制传输、批量传输、中断传输和同步传输。这 4类传输基于数据包的方式实现。而 USB的数据包分为 4种：Token、 Data、 Handshake和 Special；它们各自都有自己的数据组织方式。其中 Token令牌包只能由主机传送给设备，分为 IN、 OUT、SOF和 SETUP； SETUP包实现主机向设备发出请求 request，需要遵循USB协议特定的格式。

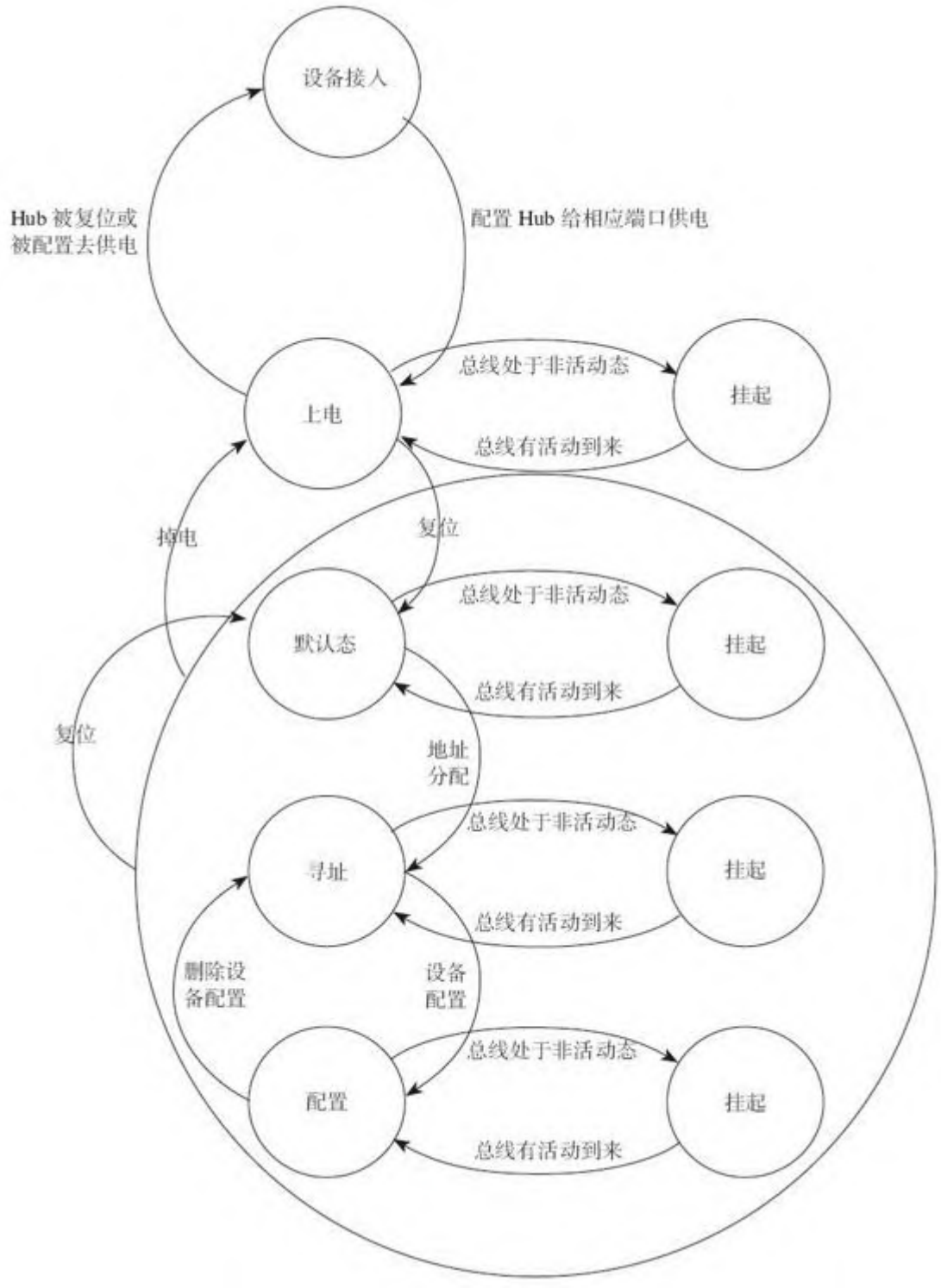


图19-4　USB启动流程

到这里关于 USB协议基础知识我们已经介绍得差不多了。这部分的内容比较粗，对于有一定 USB基础的读者来讲，可以很好带领他们复习

一下相关的知识。对于尚缺乏 USB相关知识了解的读者来讲，希望对USB有更细、更全面的了解，则可以以该节内容为引子，翻阅其他相关的 USB专业书籍。

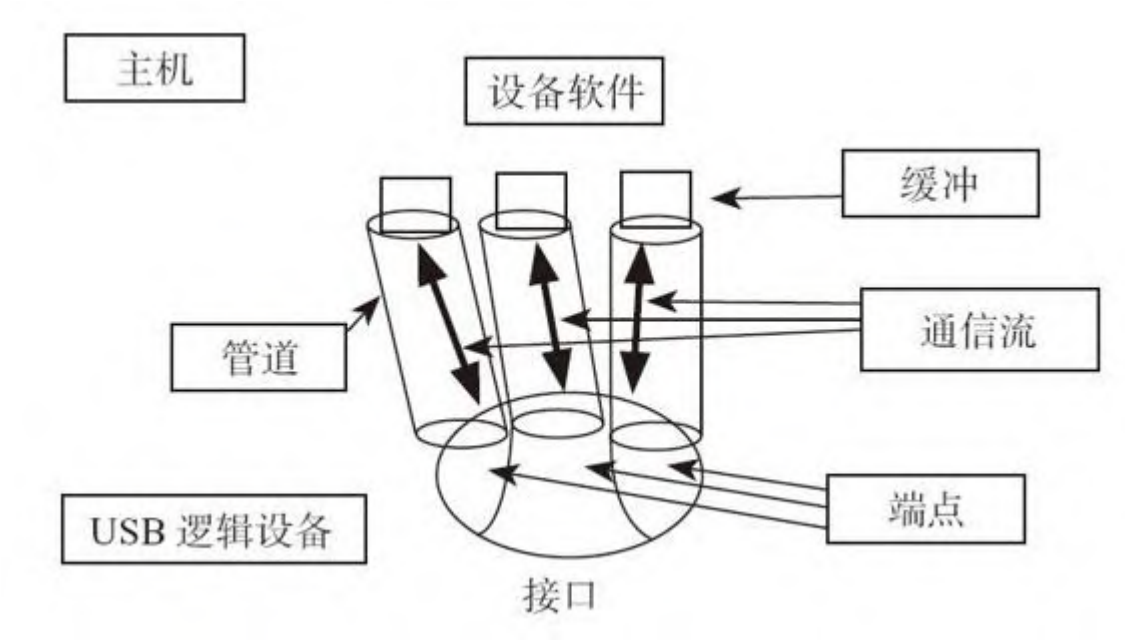


图19-5　USB管道通信

19.2 USB子系统底层

根据 19.1节的描述，我们可以知道 USB子系统，有 4类硬件组件：USB Host控制器、 USB Device控制器、 USB OTG控制器，以及 USB设备功能模块。这 4类硬件对应的就会有 4类 USB驱动。其中， OTG控制器用来决定某设备是要作为 USB Host还是 USB Device，进而启动 USB Host控制器或 USB device控制器；而 USB Host控制器则用来维护 USB通信，启动与 USB device（比如 U盘、 USB打印机等）的交互； USB Device控制器则遵循 USB通信协议，实现 USB设备与 USB Host的通信。而各 USB设备功能模块，则按照 USB设备类别，扮演好各自的功能类别，如 USB鼠标、 U盘等。

现在有的智能手机就实现了 OTG，使得其可以作为 USB Host也可以作为 USB Device。如图 19-6所示。

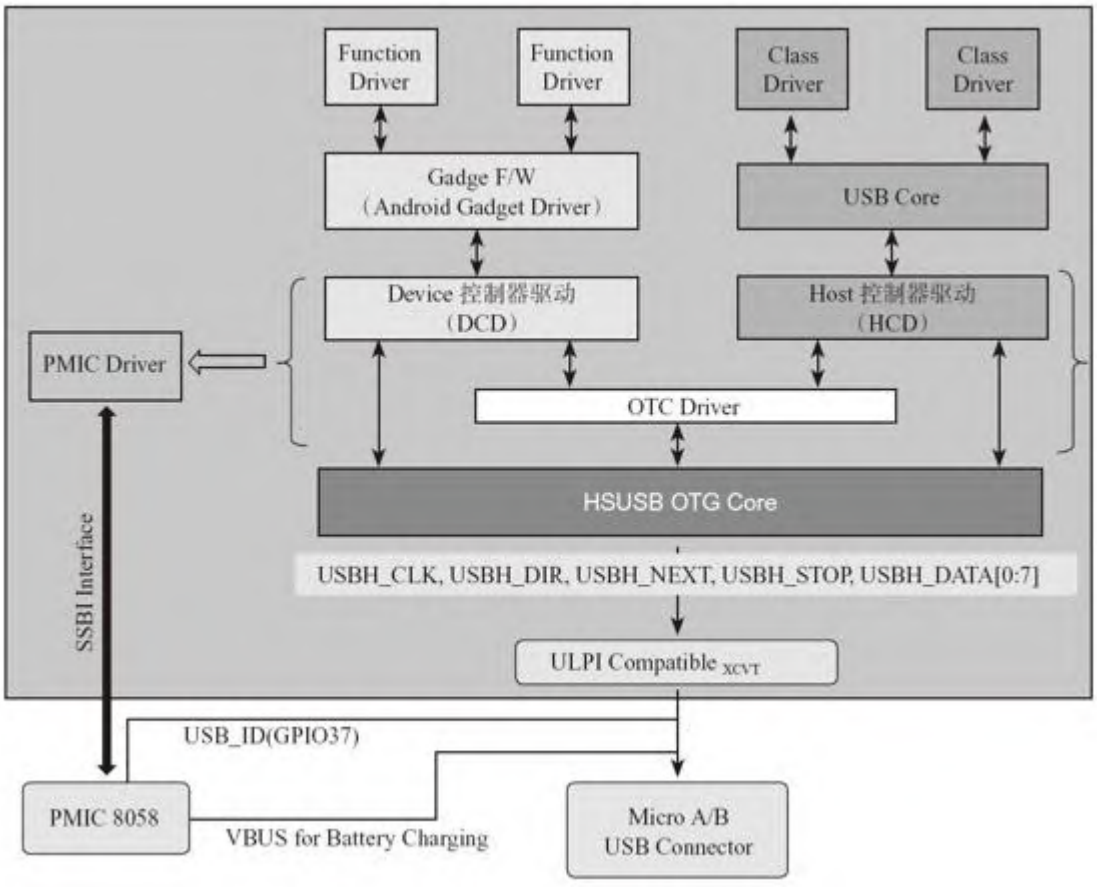


图19-6　支持USB OTG驱动栈

具体的 USB功能模块驱动种类繁多，我们将在 19.3节向大家展示一个 Mass Storage部分。下面我们就来讲讲这个 USB子系统的底层架构相关部分。

19.2.1　USB Core

Linux中的核心层已相当完善，我想一般的工程师可能不会接触它。但我们觉得多了解一点并不是坏事，至少可以加深我们对 USB子系统的理解与使用，而且说不定哪一天进了一家 CPU原厂工作，就要为相应的 CPU适配各类 USB控制器。

要理解 USB核心层，要抓住两个重点： USB总线和 urb。我们先来了解 USB子系统架构，如图 19-6所示， USB Core处在各 Class Driver与 USB HCD之间。 HCD是控制 USB Host控制器的驱动，是对 USB控制器的硬件抽象，它只对 USB负责， USB Core将各 Class Driver的请求发送到 HCD，这些 Class Driver不能直接访问 HCD，即 USB Core是 HCD与 Class Driver（也就是 USB设备）之间唯一的桥梁。

USB Core的源码位于 /drivers/usb/core目录，实现为 usbcore模块。请读者注意，它不是具体的 USB设备驱动，它是 USB设备驱动得以运行的基础，从某种意义上讲，它可以代表 USB子系统。 USB Core的初始化代码参见代码清单 19-1。

代码清单 19-1　USB Core初始化代码

```
static int __init usb_init(void)
{
    ...
    retval = usb_debugfs_init();
    ...
    retval = bus_register(&usb_bus_type);
    ...
    retval = bus_register_notifier(&usb_bus_type, &usb_bus_nb);
    ...
    retval = usb_major_init();
    ...
    retval = usb_register(&usbfs_driver);
```

```
    ...
    retval = usb_devio_init();
    ...
    retval = usbfs_init();
    ...
    retval = usb_hub_init();
    ...
    retval = usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
    ...
}
```

在该初始化函数中， usbcore注册了 usb总线、 USB文件系统、 USB Hub，以及 USB一般设备驱动 usb generic driver等。

其中 USB总线注册所用的 USB总线类型数据结构定义如代码清单 19-2所示。

代码清单 19-2　USB总线类型数据结构定义

```
struct bus_type usb_bus_type = {
.name =??"usb",
.match =?usb_device_match,?? //这是一个很重要的函数,用来匹配USB设备和
驱动
.uevent =?usb_uevent,
.pm =??&usb_bus_pm_ops,
};
```

下面我们来看看 USB设备和驱动的匹配过程。

1.USB设备驱动

前面刚讲过， USB子系统在初始化的时候就会注册 usb generic driver，其数据类型是 usb_device_driver（注意与 USB驱动类型 struct usb_driver相区别），它是 USB世界里唯一的 USB设备驱动，请参见图 19-7。

usb_generic_driver的作用可理解为：为 USB设备选择合适的配置，使得 USB设备进入 configured状态。而我们常指的 USB驱动（ usb driver），就是指 USB设备的功能接口驱动程序，比如 adb驱动程序、 U盘驱动程序等。

2.USB驱动

Linux启动时会注册 USB驱动，在 xxx_init（）调用 usb_register（）函数，将具体的 USB驱动挂载到 USB总线的驱动链表中。如图 19-8所示。

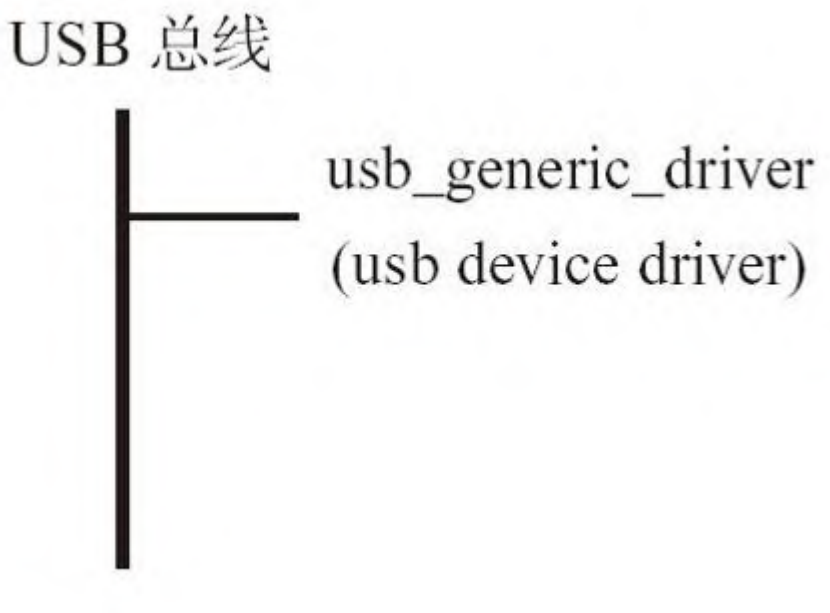


图19-7　USB总线上初始挂载的USB设备驱动

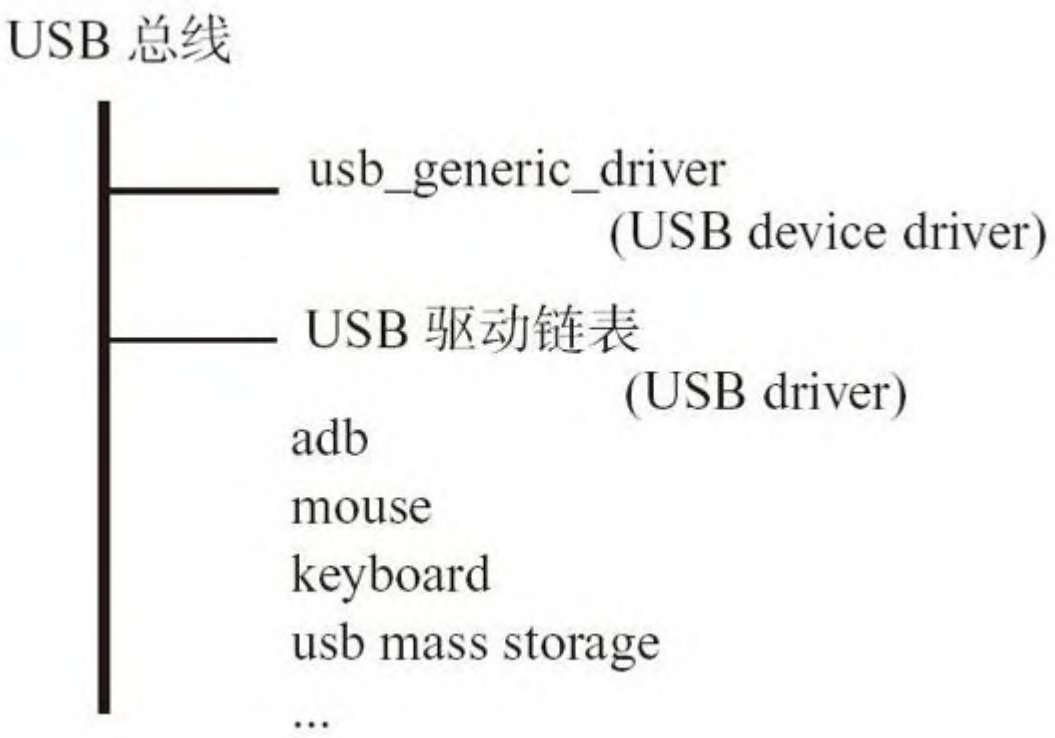


图19-8　USB驱动挂载USB总线

3.USB设备

若 USB设备连接到 USB Hub， Hub驱动检测到该设备，会为该设备分配一个 struct usb_device结构体对象，并将之挂载在 USB总线的设备链表中。如图 19-9所示。

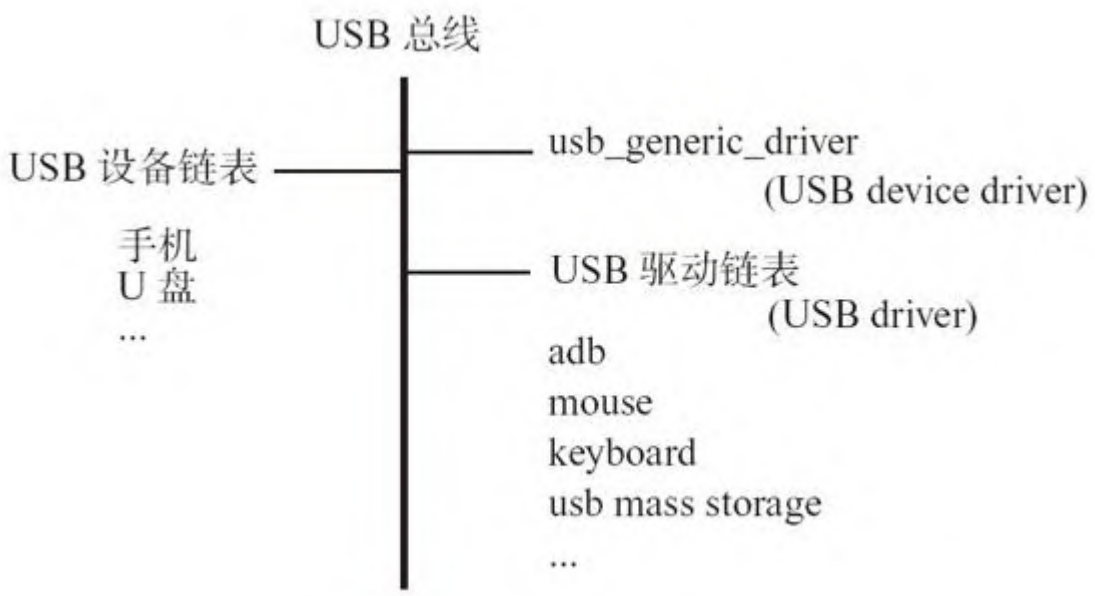


图19-9　USB设备挂载USB总线

4.USB接口

USB接口往往代表了一项 USB功能，因此该功能部分被称作 USB逻辑设备。

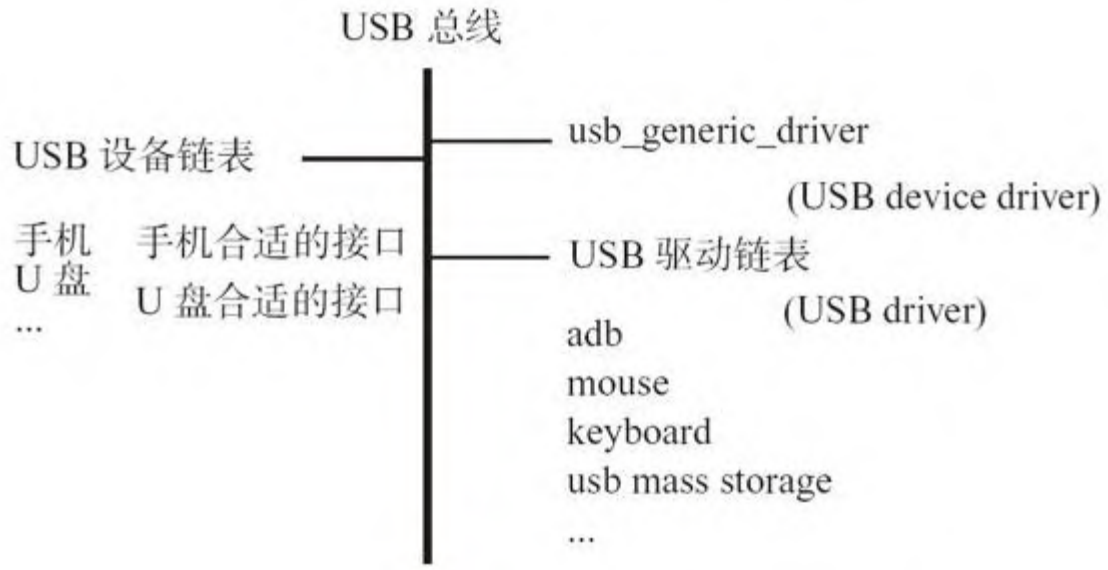


图19-10　USB设备接口（USB逻辑设备）挂载USB总线

usb_generic_driver会得到 USB设备的详细信息，然后会把准备好的接口交给 USB Core， USB Core会将该接口添加到合适的设备链表中，然后遍历 USB总线上的驱动链表，针对找到的驱动调用 USB总线的 match函数，完成 USB设备 /接口与 USB驱动的匹配。

接着，我们再来看看 USB的交互过程。

USB Host与 USB Device之间通信以数据包的形式传递， Linux遵循USB协议把数据封装成数据块以便统一调度。这个数据块就是 urb，结构体定义为 struct urb，其中的成员 unsigned char*setuo_packet就指向了前面所讲的 SETUP数据包。下面我们就来看看通过 urb完成一次完整 USB通信的过程。

1）调用 usb_alloc_urb（），以创建一个 urb，并指定 USB设备的目的端点。

2）调用 usb_control_msg（），将 urb提交给 USB Core，后者将urb交给 HCD。

3）调用 usb_parse_configuration（）， HCD解析 urb，拿到交互的数据，与 USB进行通信。

4） HCD把 urb的所有权交还给 Class Driver。

至此，我们立足于 USB Host侧，理清了基于 USB Core的 USB底层工作机制的脉络。当然 USB子系统是一个复杂的系统，更详细的剖析留给感兴趣的读者去研读代码和查阅相关的资料。下面我们进行大多数工程师所关心的 USB具体功能设备驱动程序的理解与开发。

19.2.2 Linux USB gadget三层架构

本节我们将立足于 USB Device侧，来看看 Android中 USB具体功能设备驱动架构。

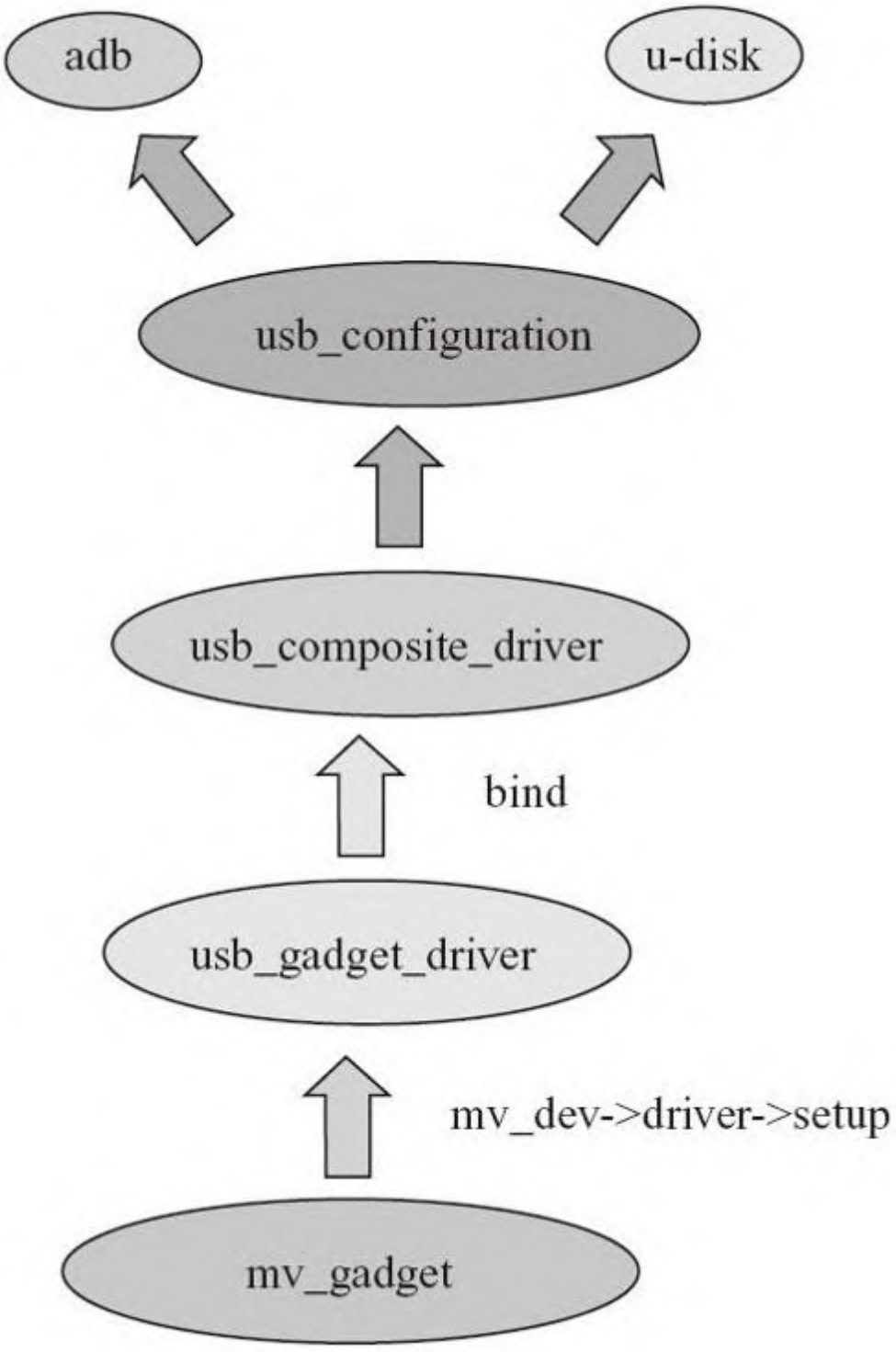


图19-11　Gadget框架实现

图 19-6就很好地展现了 USB gadget三层架构。在该三层架构中，最底层的是 USB设备控制器驱动层（ DCD），中间的就是 Gadget驱动

层，最上面的则是功能驱动层。 gadget框架从 Linux2.4.23引入以来，该框架就为 USB设备驱动开发者所喜爱并被广泛使用。该框架提出了一套标准的 API。而 USB设备控制器驱动开发者要实现这套 API；并且不同的 UDC，应该有不同的驱动，因此它是平台相关的。而Gadget驱动则实现了一套硬件无关的功能；这些功能基本可以对应到USB协议里的各 USB类（如支持通信的 CDC Class，支持鼠标键盘的 HID Class等）规范；当然该驱动所能提供的功能，还是受限于 UDC及该控制器驱动所能提供的功能。功能驱动层则实现了为大家所熟知的USB具体功能，如 U盘、 USB无线网卡等。

其实细心的读者会看到，该 Gadget驱动其实与 USB主机侧的 USB Core有那么一点神似的地方：把通用的功能抽象出来。

Android基于该框架实现了 adb、 mtp、 mass_storage等 USB设备功能驱动。 USB gadget的代码在 driver/usb/gadget目录中。三层架构在该例子的实现如图 19-11所示。

我们以 Marvell PXA910的 USB gadget来看该三层架构的具体工作机制。在该实现例子中，使用了 struct mv_usb_dev来代表其 USB设备控制器，该结构体的定义参见代码清单 19-3。

代码清单 19-3　struct mv_usb_dev数据结构定义

```
struct mv_usb_dev
{
    /*每个USB设备都有若干个端点,归属于某类USB*/
    struct usb_gadget           gadget;
    ...
    struct usb_gadget_driver    *driver;
    struct mv_usb_ep              ep[2*ARC_USB_MAX_ENDPOINTS];
    unsigned                      enabled : 1,
                                  protocol_stall : 1,
                                  got_irq : 1;
    u16                           chiprev;
    struct device               *dev;
    void*                         mv_usb_handle;
    int                           dev_no;
    u8                            vbus_gpp_no;
    ...
};
```

其中成员 struct usb_gadget用于定义 gadget；而 struct mv_usb_ep类型的数组 ep就记录着该芯片所支持的 USB端点。 struct usb_gadget_driver则对应着 Gadget功能驱动，下面我们将对它进行更详细的描述。 void*mv_usb_handle则指向对应该控制器的数据结构。

总的来讲，该控制器驱动完成了 5个工作：

1）通过 platform_data和 resource得到控制芯片的物理地址和中断号等。

2）初始化控制芯片和相关数据。

3）初始化数据 struct usb_gadget。

4）初始化数据 struct usb_ep。

5）注册 gadget服务。

再次申明， gadget设备功能驱动是指设备插入 USB主机后所具备的功能模块。像 Android手机插入 PC作 U盘，该 U盘的驱动程序就是这样一个设备功能驱动。该驱动会被挂载在 struct usb_gadget_driver下，该结构体的定义参见代码清单 19-4。

代码清单 19-4　 struct usb_gadget_driver数据结构定义

```
struct usb_gadget_driver {
    char            *function;
    enum usb_device_speed    speed;
    int             (*bind)(struct usb_gadget *);
    void            (*unbind)(struct usb_gadget *);
    int             (*setup)(struct usb_gadget *, const struct
usb_ctrlrequest *);
    void            (*disconnect)(struct usb_gadget *);
    void            (*suspend)(struct usb_gadget *);
    void            (*resume)(struct usb_gadget *);

    /* FIXME support safe rmmod */
```

```
    struct device_driver    driver;
};
```

其中， bind功能函数用来绑定 USB设备的功能，比如 U盘； setup则用来实现 USB设备的枚举。

最后，我们就从 USB设备侧来看看 USB的枚举过程。事实上，系统初始化 USB设备的过程就是枚举过程。当 USB设备插入主机，将等待来自主机的中断信号，并通过 request_irq（）进入指定的中断处理函数。设备通过读取 USB设备控制器中相应寄存器的值，得到来自 USB Host的反馈，进而执行相应的任务，枚举就是其中一项主要任务。通过调用 USB设备控制器驱动注册的枚举服务，开始枚举。该过程可归纳为以下两步：

1）通过读取 USB设备控制器，得到来自主机的请求并存放在 xxx_ctrl_req。

2）通过 switch（ xxx_ctrl_req），对具体请求做具体处理。引起处理又可分为两类：

• 标准请求处理，即对所有USB设备可通用请求的处理，如设置地址请求等。

• USB设备请求处理，即根据具体设备、具体请求的处理，其通过回调函数xxx_dev->driver->setup（）来实现。

该枚举服务回调函数 setup（）的实质，就是向 USB Host提供 USB Device的端点、接口、配置及设备各 USB描述符。而 xxx_dev->driver就是前面提到的 gadget设备驱动。根据前面获得的描述符，基于 bind函数， gadget设备驱动将为我们绑定具体的驱动功能。

19.3 Android USB子系统实践

USB是一个宏大的系统，种类繁多。为了更好地向读者展示 Android中USB子系统的实现，我们以 Mass Storage为例，其架构图如图 19-12所示。

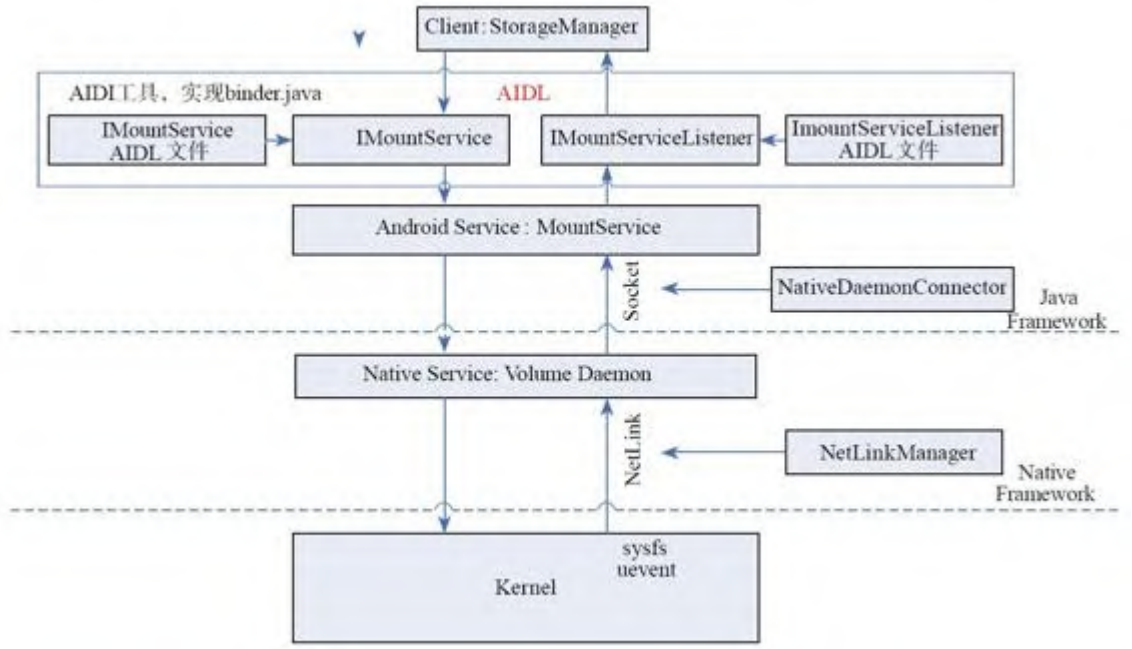


图19-12 Android USB Mass Storage架构图

19.3.1 Android IPC通信补充

在 USB Mass Storage子系统中， Android除了运用到第 18章中所讲的 Binder IPC技术，还用到了 Linux传统的 IPC技术。我觉得现在是为大家进一步理清 IPC相关概念的时候了。也只有这样，才方便我们对图 19-12做进一步的理解，对 USB Mass Storage工作流程有更深入的分析，为大家从事相关的研发工作提供扎实的知识基础。

前面讲过， IPC是指进程间的通信，就是在不同进程之间传播或交换信息。通常来讲，进程的空间相互是独立的，不能互相访问，因此也就没有办法相互交换信息了。于是 Linux OS提供了消息、同步、内存共享和远程过程调用（ RPC）等技术，来支持不同进程间交互。事实上， Linux系统空间（比如内核）、普通文件、数据库等公共资源都

可作为两个进程交互的媒介，其中前面讲的 Binder正是以内核驱动的方式实现了 Android应用程序与 Android服务的交互。所以我们应该认识到，在 Android中进程通信的手段还是挺多的。

在 Android参与通信的主体中，会常遇到一个概念 Service。而 Service在 Android中其实有 3种类型。我们应该将它们区分清楚：

• Android Application Service：它是 Android apk应用程序的 4大组件之一；它封装了一个完整的功能逻辑实现，但不像另一组件 activity运行在前台，它运行在后台；该组件常利用 Android Framework的 AIDL，通过 RPC方式来与 Android Service交互。

• Android Service：它常是一个线程，由 SystemServer启动，运行在 SystemServer进程中，它在 Java Framework层为 Android系统提供服务。

• Native Service：它也是独立的进程，由系统启动阶段解析 init.rc过程启动；它在 Native Framework层为 Android系统提供相应的服务。

有了上面的知识，我们就可以对图 19-12有一个更深的理解：Android的 USB Mass Storage实现总体上是采用了经典的 C/S架构，StorageManager就是 Client， MountService则作为一个 Android Service充当 Server的角色；对于 MountService/VolumeDaemon来讲，则又是一个 C/S架构运用，不过这次 MountService是 Client，Volume Daemon作为一个 Native Service充当 Server的角色。而 Volume Service与 Kernel的通信就不再是 C/S架构，它是通过 NetLink的方式，我们将在下一小节对这个通信展开更详细的描述。

MountService与 VolumeDaemon之间的通信采用了 Linux传统的 socket，完成了 USB Mass Storage从 Java Framework空间到 C/C++空间的转换。而 StorageManager与 MountService之间的通信是采用了 Corba的 RPC（ proxy/stub架构），如图 19-13所示。

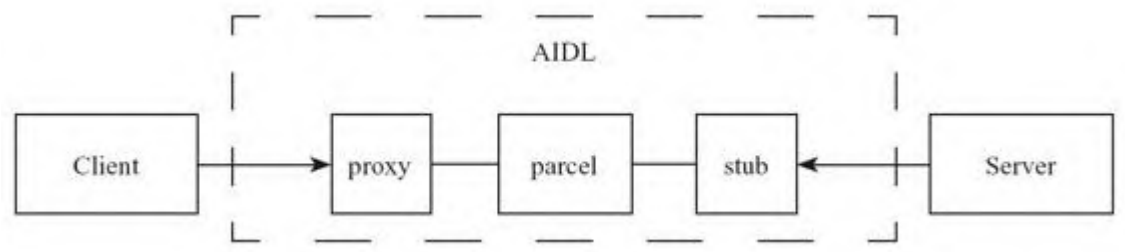


图19-13　Android proxy/stub通信架构

图 19-13中虚线框部分是 Android在 C/S架构之间加了一部分，遵循Corba组件标准的实现。愿意思考的读者也许会问，在 USB Mass Storage中，就没有用到 Binder通信机制吗？前面不是讲 Binder是Android应用程序向 AndroidService发请求的主要手段吗？这个问题问得好， Binder通信机制的运用其实就隐藏在这个 proxy/stub实现里。我们先来看代码清单 19-5。

代码清单 19-5　IMountService.java

```
/*
*这类文件是由AIDL根据Service的接口定义文件源码自动生成的,请不要修改
*该文件生成的原始接口定义文件是:
frameworks/base/core/java/android/os/storage/IMountService.aidl
*/
packageandroid.os.storage;
/**注意! 如果这个文件有修改,还应确保对应的IMountService.h与
IMountService.cpp两个文件有修改
*特别是,下面的方法必须与IMountService.cpp文件中的TRANSACTION_xxx枚举
相一致
* @hide -上层应用应该使这些android.os.storage.StorageManager定义的
API来访问storage相关的功能

*/
public interface IMountService extends android.os.IInterface
{
    /*本地端IPC实现stub class. */
    public static abstract class Stub extends android.os.Binder
implements android.os.storage.IMountService
    {
        @Override public boolean onTransact(int code,
android.os.Parcel data, android.os.Parcel reply, int flags)
```

```
throws android.os.RemoteException
        {
            switch (code)
            {
                ...
                case TRANSACTION_registerListener:
                {
                    data.enforceInterface(DESCRIPTOR);
                    android.os.storage.IMountServiceListener
_arg0;
                    _arg0 =
android.os.storage.IMountServiceListener.Stub.asInterface(data.
readStrongBinder());
                    this.registerListener(_arg0);
                    reply.writeNoException();
                    return true;
                }
                ...
                return super.onTransact(code, data, reply,
flags);
            }

            private static class Proxy implements
android.os.storage.IMountService
            {
                ...
                /*为异步接收notifications而注册一个IMount
ServiceListener*/
                public void
registerListener(android.os.storage.IMountServiceListener
listener) throws android.os. RemoteException
                {
                    android.os.Parcel _data =
android.os.Parcel.obtain();
                    android.os.Parcel _reply =
android.os.Parcel.obtain();
                    try {
                        _data.writeInterfaceToken(DESCRIPTOR);

_data.writeStrongBinder((((listener!=null))?
(listener.asBinder()):(null)));

mRemote.transact(Stub.TRANSACTION_registerListener, _data,
_reply, 0);
                        _reply.readException();
                    }
                    finally {
```

```
                    _reply.recycle();
                    _data.recycle();
                }
            }
            ...
        }
}
```

该文件是向外提供访问 MountService这个组件服务遵循 Android规范的接口。从上面的源码我们可以看到， IMountService中的 Stub继承 Binder和 IMountService。而我们可从源码中找到与图 19-13的对应：

- StorageManager - >Client
- IMountService.Stub.Proxy->Proxy
- IMountService.Stub->Stub
- MountService->Server

图 19-13中间还有一个 Parcel，是指将要通信的对象序列化，也就是为了满足 RPC规范而执行的打包动作，等同于微软 COM规范中的 Marshaling。

当 StorageManager想要调用 MountService方法时（这要一次 Android的 IPC通信），例如调用 registerListener，步骤如下：

1）进入 IMountService.Stub.Proxy找到方法 registerListener。

2） IMountService.Stub.Proxy.registerListener利用 Parcel将函数调用序列化为 Android所理解的结构。

3）调用 onTransact（）函数，该函数会根据参数，找到 registerListener switch-case语句执行。

4）数据通过 Binder机制进行写操作，客户侧调用阻塞，等待服务器侧应答。

5）服务器处理完 request返回。

6）客户端取回数据。

到这里，读者是不是对 Android的 IPC，特别是 Binder通信机制有了更深入的认识，同时对 Binder这个通信机制为什么比消息等 IPC通信技术更轻便快捷有了更好的理解？

19.3.2　Android USB Mass Storage流程分析

19.3.1节我们对 Android USB Mass Storage的通信有了较深入的讲解，现在我们来讨论 Android USB Mass Storage的工作机制。并以此为例子，希望对大家关于 Android USB子系统全面的学习与理解有所帮助。我们讨论将围绕图 19-12中的 Volume Daemon展开。

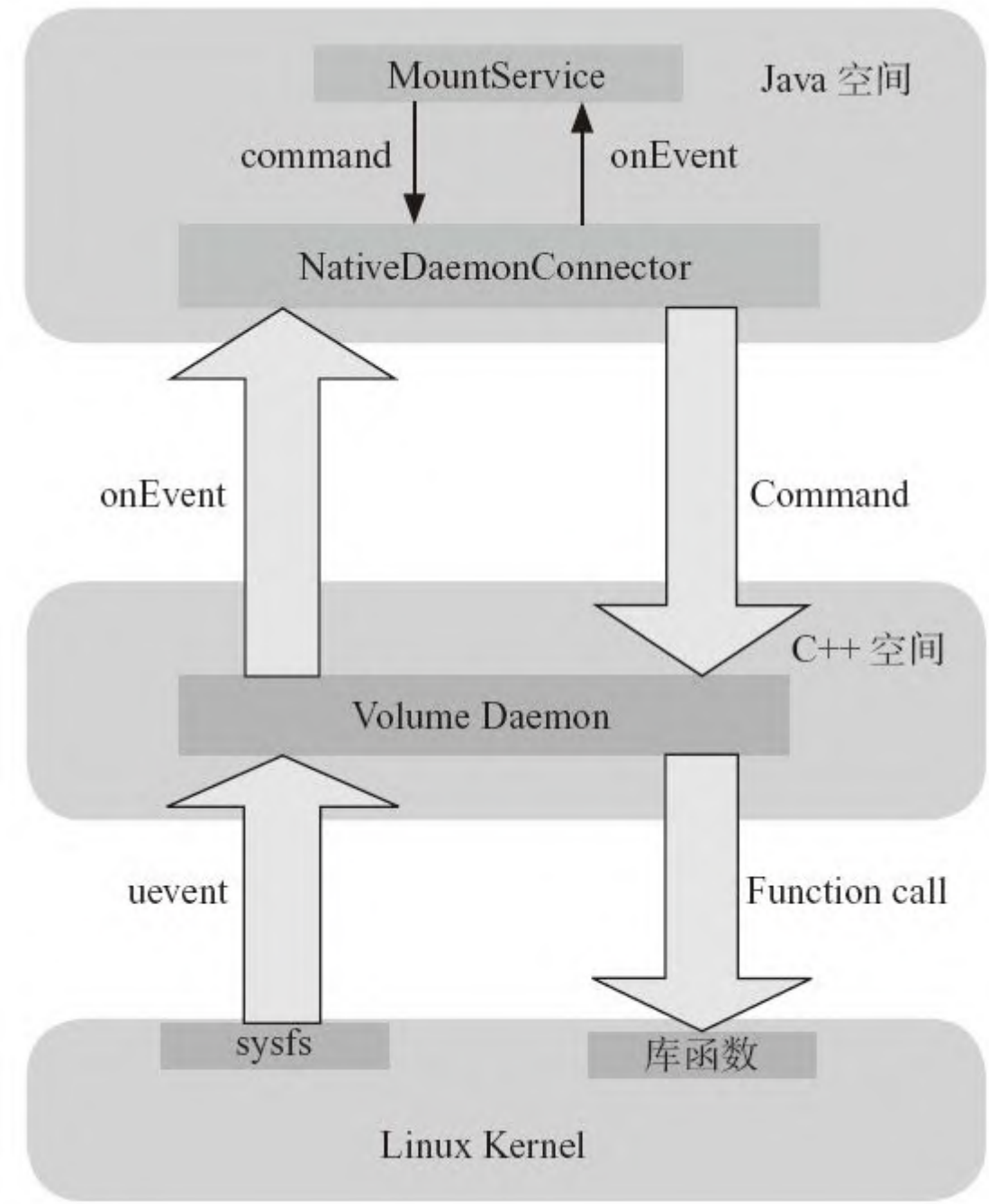


图19-14　Vold工作机制

Volume Daemon是一个 Native Service，它的简称是 Vold，用来管理USB/sd卡等外部存储设备，这里我们主要关注 USB。从图 19-12中我们也可以看到，它是通过内核中 sysfs uevent机制，来获知新驱动加载的 event。关于 Vold的主要工作机制如图 19-14所示。

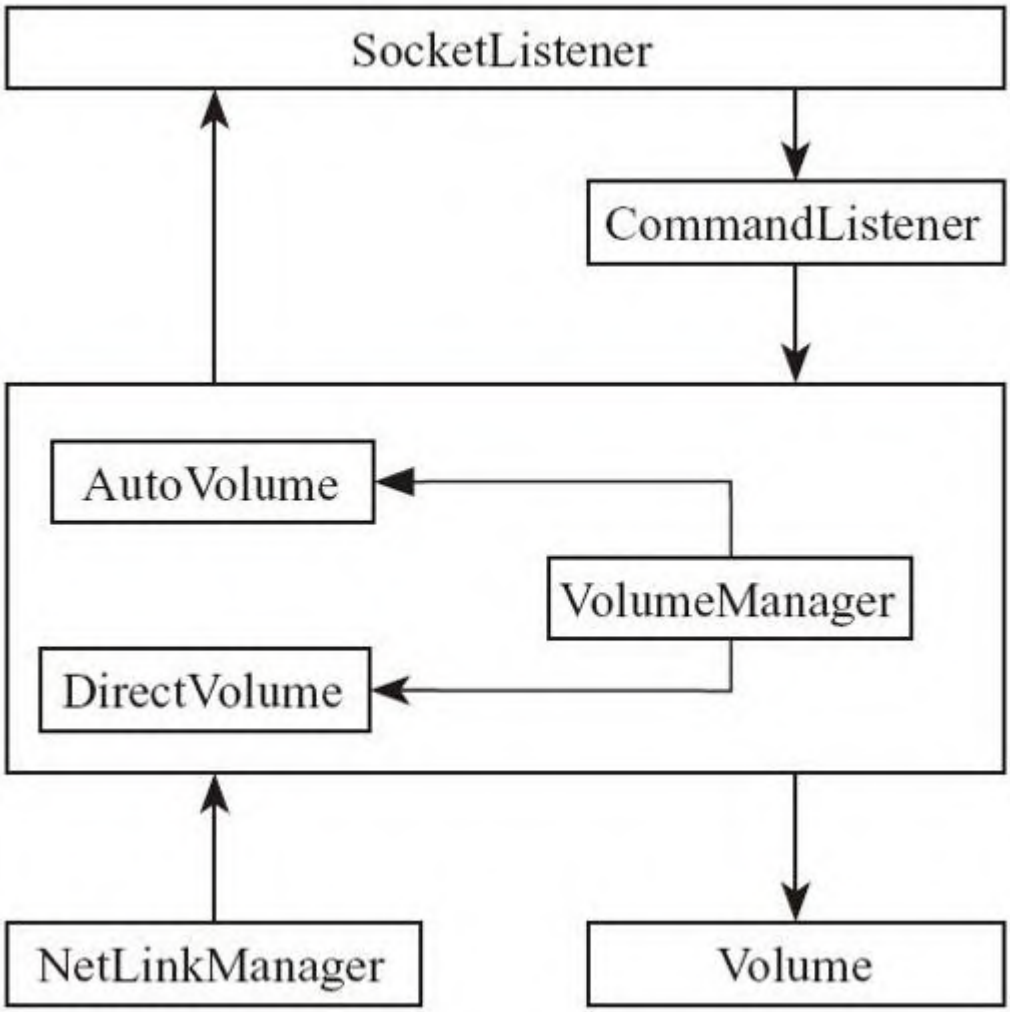


图19-15　Vold内部框架

Vold在启动时会创建一个 socket，以用于 Vold与上层 Framework的通信。在 Vold的 main函数中，还会创建 VolumeManager和 NetLinkManager两个类的实例，其中后者就会创建 NetLink Socket。NetLink则负责 Vold与 Linux内核通信。这样， Vold就与 Kernel、Framework的通信框架搭建好了。而 Vold自身内部框架如图 19-15所示。

总地来讲， Vold工作处理流程主要有 3步：创建连接、引导和事件处理。其中创建连接就是 Vold作为中间层建立与底层内核和上层 Framework的通信。这步上面已讲过，在 Vold的启动过程中就完成了。而引导就是在 Vold启动时，就对 Android设备开机就有的外存储

设备（ SD卡常存在这种情况）进行处理。最后事件处理，就是在Vold的运行期间，基于上面建立与底层和上层的连接，处理 Kernel发出的 uevent或上层 Frame work发送的命令。

当 NetLinkManager检测到 Kernel发出的 uevent（比如 U盘插入），就会调用 NetLinkHandler::onEvent（）处理；当是 "block"事件时，由于多态性最终选择 AutoVolume或 DirectVolume来处理。而当CommandListener检测到 Framework层命令，就会遍历 VolumeManager中的 Volume清单，找出对应 Volume，调用 Volume函数；而 Volume的功能，最终由 Linux内核落实。关于NetLinkHandler::onEvent（）的代码参见代码清单 19-6。

代码清单 19-6　NetlinkHandler::onEvent（）

```
Void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    ...
    VolumeManager *vm = VolumeManager::Instance()?;
    const char *subsys = evt->getSubsystem()?;
    if(!subsys) {
        ...
    }
    if(!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt)?;
    } else if (!strcmp(subsys, "switch")) {
        vm->handleSwitchEvent(evt)?;
    } else if(...) {
        ...
    }
}
```

从 19.2节中我们了解到，当 USB Mass Storage设备被枚举时，在Linux内核底层就会完成相应 USB设备与驱动的关联。而 Android系统要做的就是如何获取该设备的事件，以及如何向上层应用展现该设备的功能。下面我们就来分析 Android系统中关于 USB Mass Storage工作原理，如图 19-16所示。

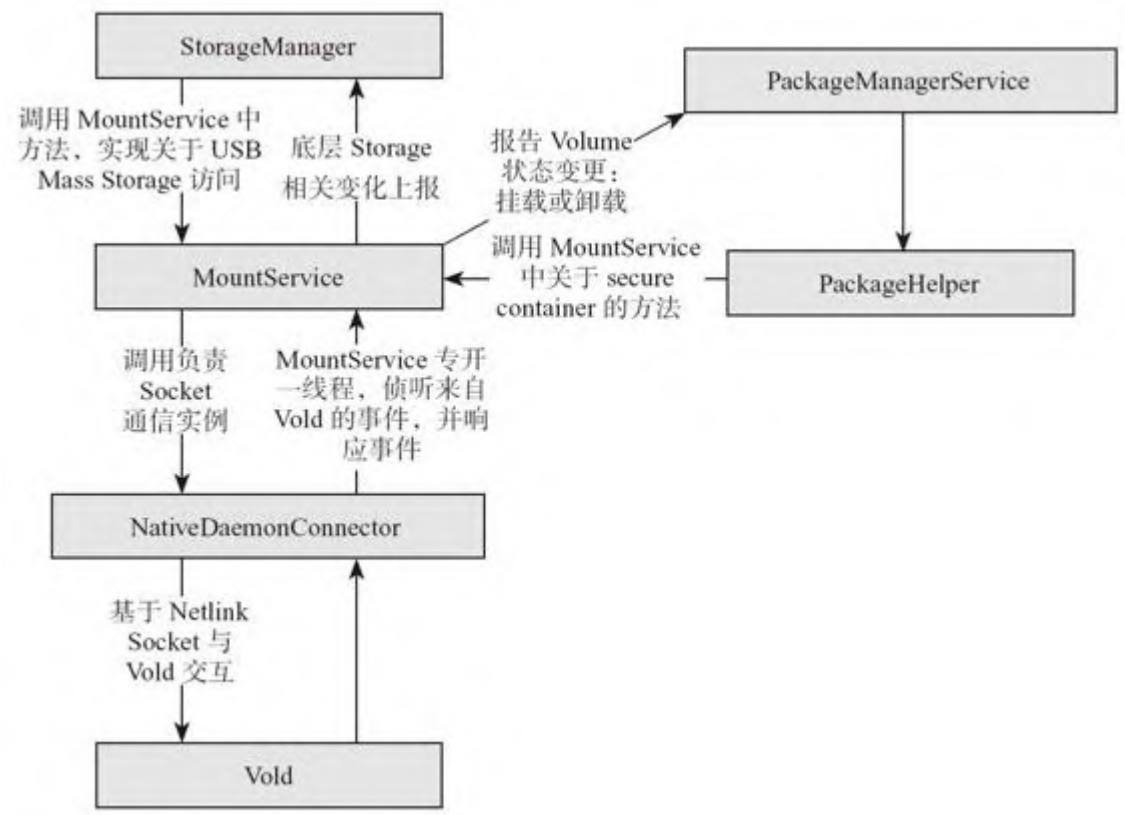


图19-16　USB Mass Storage工作交互

如图 19-16所示， MountService用于管理 USB存储设备的后台服务，实现对存储设备的操作。应用程序则通过 StorageManager来访问该服务。 StorageManager向应用程序提供了相应的方法，如查看 USB Mass Storage状态方法、设置 USB Mass Storage Enabled/Disabled等。

PackageManagerService用于管理 Android系统中所有 apk，当然关于USB Mass Storage的应用也不例外；而且这些 apk的安装文件常在Mass Storage中，安装所生成的文件也要存放在 Mass Storage上。图19-16中 PackageHelper是一个辅助类，它可以帮助PackageManagerService调用 MountService中的方法。最后，NativeDaemonConnector会获得 Vold创建的 socket，实现MountService与 Vold的通信。

NativeDaemonConnector运行的主体是一个线程。该线程会通过上面所讲的 socket来监听来自 Vold的消息；这些消息包括：VolumeStateChange、 ShareAvailabilityChange、

VolumeDiskInserted、 VolumeDiskRemoved、 VolumeBadRemoval；接收到消息后会调用 onEvent做处理。 MountService也能通过调用 doCommand向 Vold发送命令，进而访问底下的 USB Mass Storage的内容。

这里我们关于 USB子系统的讲解就接近尾声了。关于 USB子系统中的其他功能部分，比如 USB键盘，读者可以参照 USB Mass Storage，在 Android相应的子系统找到相应代码与文档，比如 Input子系统。

第20章　Bootloader引导子系统

Bootloader也是Android系统的重要组成部分，它与Linux内核、Android系统一道，在Android编译过程中，会生成各自对应的Image文件。这三个Image文件就是针对Android设备打包好的三大必要软件系统。

Bootloader负责Android机器的启动、Image的烧写、关机充电等，这些都是与内核要实现的关键器件的驱动分不开的。当然由于考虑系统启动资源更加有限，以及尽量加快机器启动要求，这些驱动常是仅实现最精简的功能。因此在Android中，Bootloader又有一个简称LK（Little Kernel）。

20.1　Bootloader流程分析

本节我们将主要讲述 Android引导子系统 Bootloader的关键动作，至于启动中更详细的过程，最好是研读具体 Bootloader代码。相信通过介绍 Bootloader的流程框架，大家对 Android的开机过程会有一个更清晰的认识。

20.1.1　Bootloader概述

Bootloader是操作系统运行之前运行的一段程序，它最主要的工作就是将机器的软硬件环境带到一个适当的状态，为操作系统的运行做好准备。具体讲， Bootloader要完成的子任务很多。但这些任务的目的总体说就是要努力把 OS拉出来，在机器的 CPU等硬件组成的环境上运行。

在嵌入式系统领域里， Bootloader的种类繁多。 ARM处理器系列的 Bootloader则大多数以 U-Boot为基础。 U-Boot是一个开源的项目，大家可以从网上下载到它的最新源码。

为了节省成本和增加 Flash存储容量，现在嵌入式系统都会使用 Nand Flash作为操作系统和程序存储体。但 Nand Flash上不能直接运行程序，因此通过它难以做到上电就启动程序，维持机器的正常运转。为此现在处理器都会集成 BootROM，这类存储可以直接存放可执行程序，以及被 CPU直接访问数据。处理器芯片厂商会在其上烧写初始化硬件和提供下载功能等程序，这些程序在机器上电后就会得以执行。在一般正常开机过程中，就会将 Nand Flash中 Bootloader加载进 RAM，并将 CPU指令计数器（ PC）指向 Bootloader程序的起始地址。

往往在 Bootloader程序中，除了会有 U-Boot程序外，还会有特定于某处理器芯片的处理代码。比如 BroadCom BCM2157的启动代码顺序其实是： BootROM->boot1/boot2->U-Boot，如图 20-1所示。

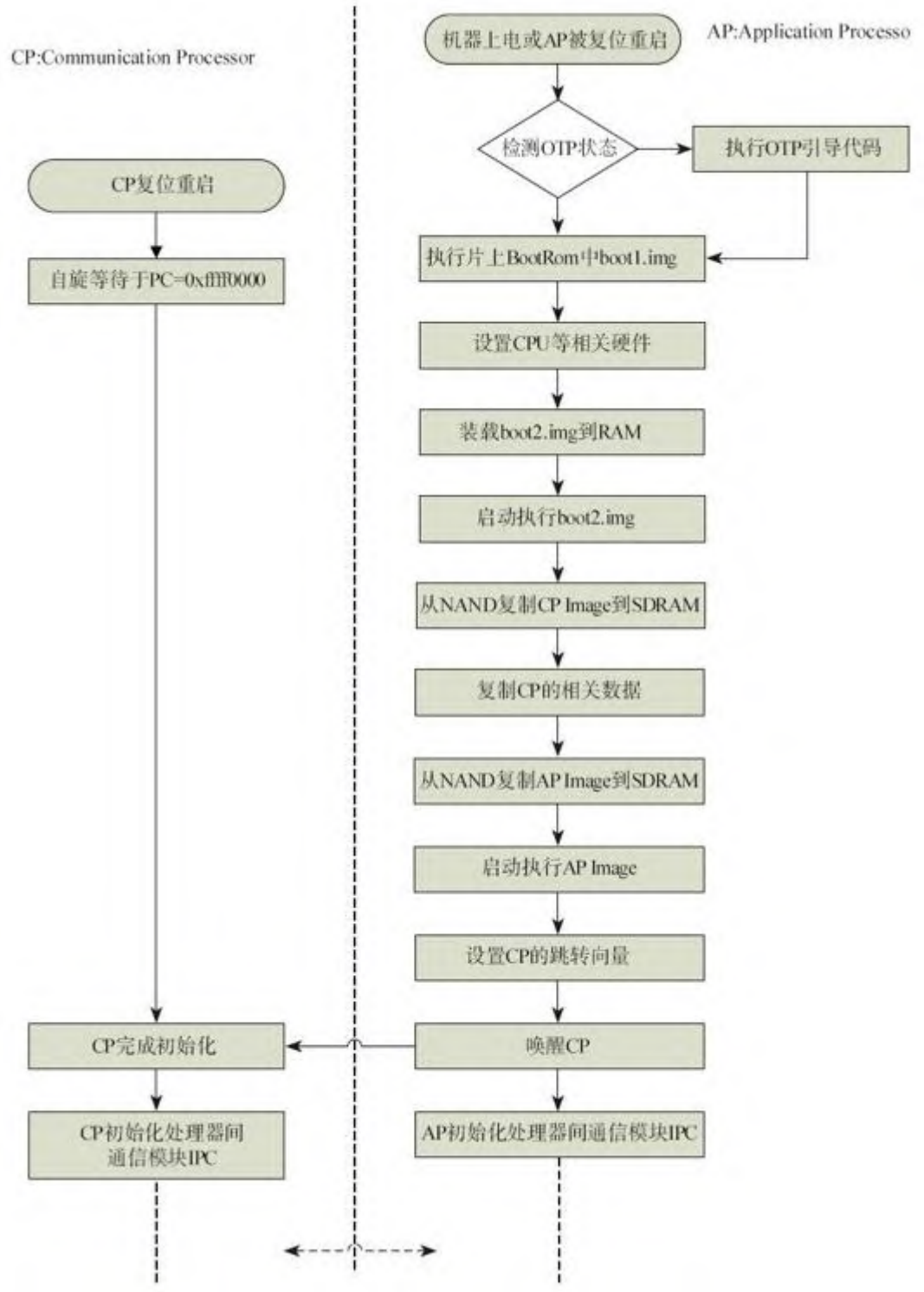


图20-1　BCM2157 Android BootLoader启动流程

特定于某平台的启动代码与流程我们这里不讲，下面就来看看为大多数 Android平台所通用的 U-Boot。

20.1.2 U-Boot启动流程分析

现在 Android手机都会有两个处理器：应用处理器 AP，负责主操作系统、与用户交互应用程序的执行；通信处理器 CP，负责无线电话通信功能，并通过 AP向 Android终端用户提供相应的无线通信功能。在 U-Boot启动中，这两类处理器都会涉及。但由于 CP相对稳定，而且这部分工作复杂而机密，一般都由芯片厂商适配好了，所以我们不关心 CP的启动，而主要关心 AP侧的。

从代码执行顺序和代码编写语言来分， U-Boot的启动过程可分成两个阶段：第一阶段，常称汇编代码阶段，在这个阶段由于要考虑存储空间受限和启动速度要尽量快，所以其代码编写采用效率更高的汇编语言；第二阶段， C代码阶段，到此阶段已可以开始与用户交互，而且机器的硬件大多已激活，所以一般采用 C语言编写。

在 Android Bootloader中，第一阶段的代码实现在 start.S中。在该段代码里， U-Boot要完成 DDR、 Nand和 Nand控制器等基础硬件的初始化。就 ARM处理器而言，具体要做的工作如下：

1）设置 CPU进入 SVC（系统管理模式）， cpsr[4:0]=0xd3。

2）关中断， INTMSK=0xffffffff， INTSUBMSK=0x3ff。

3）关看门狗， WTCON=0x0。

4）调用 xxx_cache_flush_all函数，使得 CPU上的 TLBS、ICache/DCache、 WB中数据失效。

5）设置 CPU当前工作时钟。

6）检查系统的复位状态并记录，以确定后面要执行何种开机流程（正常开机、插充电器开机、按复位键开机、执行复位命令开机、进入recovery模式开机、睡眠后唤醒开机等）。

7）关闭 MMU，打开 Icache和 Fault checking，并调用 memsetup.S中的 memsetup函数来建立对 RAM的访问时序。

8）调用 relocate函数，加载 Nand Flash中其他 U-Boot代码到 SRAM中，并执行。

最后一步的流程要完成的任务较多，如图 20-2所示。

事实上， start_armboot（）函数实现在 board.c中，因此至此就进入了 U-Boot的 C代码阶段。 U-Boot第二阶段主体工作顺序如下：

1）定义 struct global_data结构体指针 gd，该指针存放在 CPU的寄存器中；另外还定义了全局结构体对象 struct global_data gd_data、 struct bd_info bd_data，以及指向函数的二级指针 init_fnc_ptr。

2）将指针 gd指向 gd_data， gd->bd=&db_data；将 gd_data其他成员清零。

3）初始化函数指针数组 init_fnc_ptr为 init_sequence。接着会有一个 for循环，其会提取该数组中的每个函数依次执行，以完成 CPU与板级各 U-Boot所需基本功能模块的初始化。下面是一份该数组的代码清单 20-1。

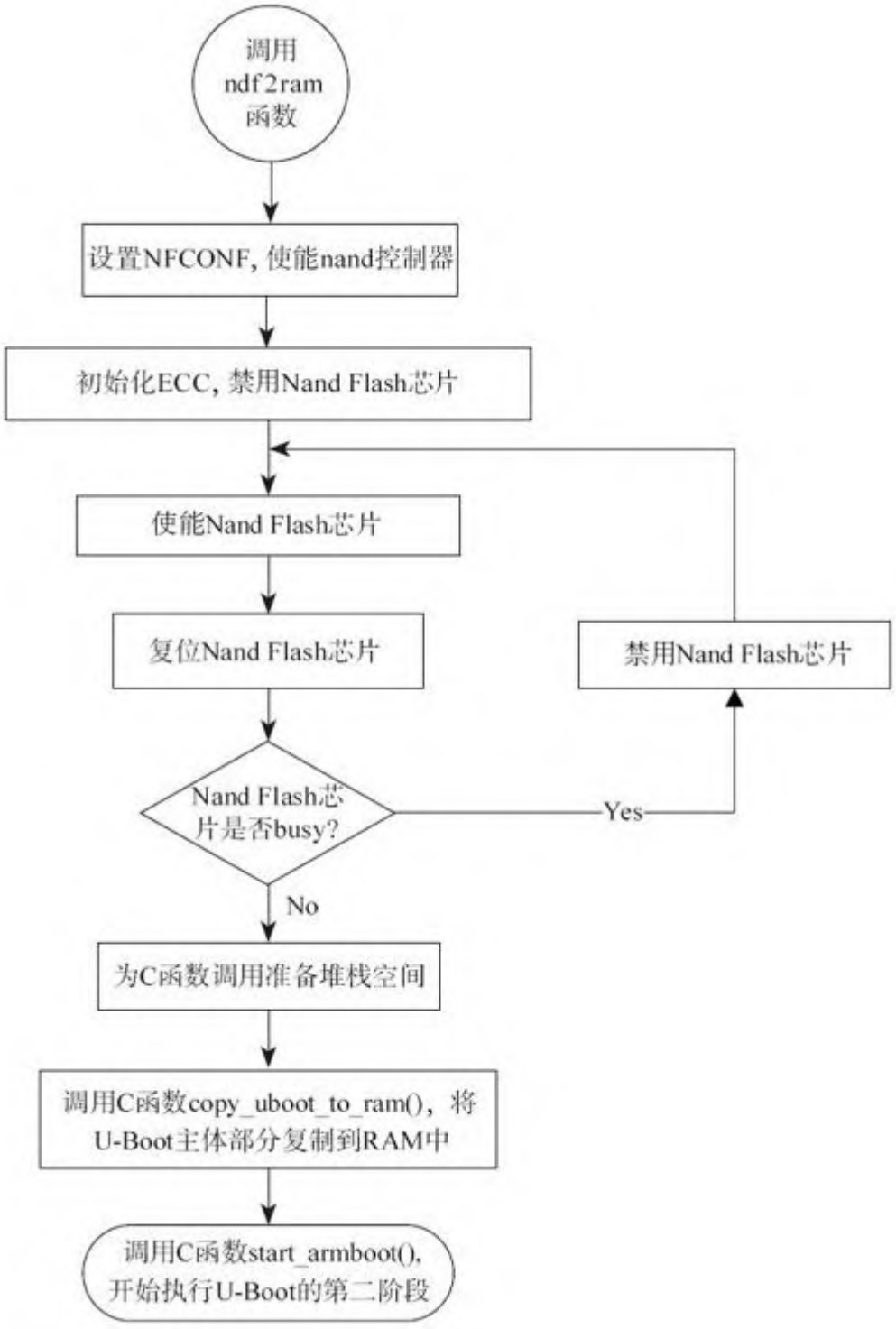


图20-2　relocate函数处理流程

代码清单 20-1　 init_sequence[]

```
init_fnc_t *init_sequence[] = {
    cpu_init,        /*基本的处理器相关配置, 根据需要设定IRQ、FIR堆栈
*/
    board_init,      /*基本的板级相关配置,设置LOCKTIME,配置MPLL、
UPLL,配置IP ports,设置gd->bd->bi_arch_number, gd->bd-
>bi_boot_params,使能ICache和DCache等 */
    interrupt_init,/*初始化中断处理,设置系统时钟滴答等,但此时中断尚未打
开,中断还不会响应 */
    env_init,        /*初始化环境变量,这些环境变量可以保存在内存,但对于
掉电后仍要保存的参数,则应保存在flash上 */
    init_baudrate,  /*初始化波特率设置 */
    serial_init,     /*串口通信设置 */
    console_init_f, /*控制台初始化阶段1 */
    display_banner, /*打印U-Boot信息,如版本号、bss开始地址、IRQ堆栈地
址等 */
    dram_init,       /*配置可用的RAM */
    display_dram_config,/*显示RAM的配置大小 */
#if defined(CONFIG_VCMA9)
         checkboard,   /* display board info */
#endif
    NULL,
};
```

4）配置可用的 flash空间，并打印出相关信息，由功能函数 flash_init（）和 display_flash_config（）分别完成这两个任务。

5）调用 mem_malloc_init（）函数分配堆空间。

6）调用 env_relocate（）函数，将机器环境参数复制到上面分配的堆空间中。

7）调用环境变量列表中的 ipaddr参数、 MAC地址，并设置好机器的以太网相关的参数。

8）调用 devices_init函数，创建 devlist，但该设备列表中目前只有一个串口。

9）调用 console_init_r（）函数，完成控制台的最终初始化。至此，我们可以调用 serial_getc（）、 serial_putc（）、 putc（）和 getc（）等功能函数，来获取 U-Boot相关的调试信息。

10）使能中断。

11）如果要用网卡设备，设置好网卡的 MAC和 IP地址。

12）调用 main_loop（）函数。该函数将完成 U-Boot的初始化工作。该函数将接收标准输入设备的输入，根据输入的不同将执行不同的开机流程。 main_loop（）函数主体代码参见代码清单 20-2。

代码清单 20-2　main_loop（）主体代码

```
void main_loop()
{
    ...

    s = getenv ("bootdelay");   //自动启动内核等待延时
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) :
CONFIG_BOOTDELAY;
    s = getenv ("bootcmd");     //取得环境中设置的启动命令行
    if (bootdelay >= 0 && s && !abortboot (bootdelay)){
        run_command (s, 0);     //执行启动命令行
    }
        for (;;) {
            len = readline(CFG_PROMPT); //读取键入的命令行到
console_buffer
            flag = 0;       /* assume no special flags for now
*/
            if (len > 0)
                strcpy (lastcommand, console_buffer); //复制命令
行到lastcommand
        else if (len == 0)
            flag |= CMD_FLAG_REPEAT;
        if (len == -1)
            puts ("\n");
        else
            rc = run_command (lastcommand, flag);  //执行这个命令
行。
        if (rc <= 0) {
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
}
```

对于一般用户而言，不会输入什么命令。此时， main_loop（）函数的实际执行流程是：接收开发者所配置的 bootcmd环境，直接加载Linux内核 Image到机器 RAM中，并将 CPU的控制权交给新加载的Linux内核。

20.2 Bootloader修改指南

通过前面的学习，我们了解了 BootLoader流程，理解了 BootLoader就是一个顺序执行的程序（它没有进程或线程切换的概念）。但作为机器初始启动的程序，它的有些作用是 Linux操作系统所不能具备的。这些功能有：

- 开机上电的第一帧图。

- 特殊的按键组合，使得机器进入fastboot模式，以支持Android系统各Images的烧写、升级。

- 启动模式的判断、分析与响应。

正因为如此，许多厂商与运营商在 BootLoader中封装它们定制化专有的东西。例如，联通 Android手机都会要求在这里加上联通相应的徽标，而且不允许用户通过升级程序将之删除。下面我们就来看看这些功能具体如何实现。

20.2.1 开机第一帧图的修改

我们要记住， Android机器在启动后，可显示的第一帧图不是由Linux提供的，更不是经常看到的 Android开机动画，而是在BootLoader中加载的 Splash画面。

为了在 BootLoader中显示图像，需要将 framebuffer驱动及 LCM的display驱动集成进 BootLoader中。而图像数据的存储方式主体上有两种：一种是将要显示的图像数据存放在 Flash存储器上，另一种则是将图像数据转换成数组隐含在 BootLoader程序中。这两种方式各有优劣点，不同的厂商可能会选择不同的方式。所以在 BootLoader中修改要显示的第一帧 Splash图片，要查看相应厂商的参考代码。

作为一般开发人员，修改工作主要包括：

1）找到 LCM display驱动代码所在的地方。

2）针对自己所选 LCM，编写合适的显示屏驱动。

3）查看代码编译脚本，将自己的 LCM驱动替换原厂参考代码中的 LCM显示驱动。

4）不用担心 Framebuffer驱动，因为原厂已在参考代码中适配好。

5）找到 Splash图像数据的存储方式。

6）根据存储方式准备自己的图像数据，并替换参考代码中的数据。

这部分的功能代码厂商往往会用宏来控制与关闭，比如高通公司在MSM系列芯片中就用到了宏 DISPLAY_SPLASH_SCREEN。根据这些宏很容易找到想要修改的代码与数据。下面就以 MTK的 mt65xx平台看看这部分功能修改所要牵涉的内容。

LCM驱动所在的目录为：
bootable/bootloader/lk/custom/full/kernel/lcm。针对不同的 LCM驱动芯片在该目录下都有一个子目录，比如 ili9481；而在该子目录就存放着对应的 LCM显示程序文件，比如 ili9481.c。在bootable/bootloader/lk/custom/full/kernel/lcm中增加自己要适配的 LCM显示驱动之后，要修改该目录下的文件：
mt65xx_lcm_list.c，以便 BootLoader中能用到刚适配好的 LCM驱动。而且我们应该定义一个宏，比如： ILI9481，以便我们从众多的LCM驱动中选取我们想要的。

关于上电启动显示的 Splash图，在 mt65xx平台中选择了一个 Image —— logo.bin来存放这些图像资源，而这个 Image将被烧写在独立 nand flash分区（ LOGO分区）中。作为开发者来讲，就是要在该目录下定义一个子目录，在该目录下存放想在 BootLoader中显示的第一帧图的位图文件，并定义 BOOT_LOGO指向新建的子目录。之后进行编译， MTK的编译环境就是调用该目录下 tool子目录的 bmp_to_raw与 zpipe，将位图资源解析成 raw数据，同时将这些 raw数据打包进 logo.bin Image。而在机器上电启动时， Bootloader会在调用 platform_init（）函数中显示所要的开机 Splash图像。

20.2.2 开机模式的定制

事实上， BootLoader实现的关键就是对机器上电开机模式的判断、分析与响应。前面讲过， Android机器开机重启，运行到 BootLoader的

原因有很多种。而有的原厂所给的参考代码中，没有对某种启动原因的判断或对某种启动有很好的响应处理。像笔者就发现有的代码中就没有针对充电开机的处理，所以在此种情形下，机器是傻乎乎地直接从 BootLoader启动到 Linux，再在 Linux电源管理系统运行后，利用Linux睡眠唤醒机制处理这类充电开机启动，结果是用户开机体验很不好（因为要等较长的时间开机启动 Linux， Linux运行在合适的阶段才能判断是充电开机，还是正常的睡眠与唤醒，那在短时间里用户其实很难知道机器是否正常响应了）。如果基于开机模式的定制，往往可以获得一些我们意想不到的独有机器特性。所以基于原厂的BootLoader参考代码，进行开机定制方面的修改还是有必要的。

要想掌握定制这方面的内容，根据我们的经验就是要研读相关的BootLoader代码。因为这块平台相关性很强，不同的 CPU、不同的硬件板子，就可能意味着不同。通过研读代码，以明白两件事件：

1）如何判断开机的模式？

2）针对不同的开机，应有何种处理响应？

在 BootLoader代码中，会定义一个全局变量 g_boot_mode来记录开机模式。我们还是以 MTK的 mt65xx为例看开机模式的判定。我们可以看到，在进入 BootLoader之后，会调用 boot_mode_select（）函数，而该函数将会读取 CPU寄存器、特殊的管脚、按键的组合等来判断机器启动原因，并将这些原因记录在变量 g_boot_mode中。从参考代码中，我们可以看到，对以下的开机原因做了处理： META_BOOT、 FACTORY_BOOT、 RECOVERY_BOOT、 SW_REBOOT、 NORMAL_BOOT、 ADVMETA_BOOT、 ATE_FACTORY_BOOT、 ALARM_BOOT、 FASTBOOT与 UNKOWN BOOT。

UNKOWN BOOT就代表未知的开机原因。从上面列举来看，没有看到与充电开机相关的处理，换句话说，充电开机是不是就当 UNKOWN BOOT处理了？如果真是这样，机器在充电开机的情况下，将不能正常开机。为了保证充电开机下能正常开机，将变量 g_boot_mode默认设置值为NORMAL_BOOT。但这种处理其实在 BootLoader中对充电开机基本不作为，结果导致充电图标明显出现得慢、关机充电比睡眠充电慢等问题。

为此，我们针对充电开机做了如下修改：

1）在枚举数据 BOOTMODE中增加 CHARGING_BOOT成员。

2） boot_mode_select（）函数中增加 CHARGING_BOOT检测分支。参见代码清单 20-3。

代码清单 20-3　修改过的 boot_mode_select函数

```
void boot_mode_select(void)
{
    ...
    #ifdef CFG_RECOVERY_MODE
    if(recovery_detection())
    {
        ...
    return;
    }
    #endif
        #ifdef CFG_RECOVERY_MODE
    if(charging_detection())
    {
        return;
    }
        #endif
}
```

而在获得充电开机的原因后，我们可以很好地在 BootLoader中对充电开机进行处理，如图 20-3所示。

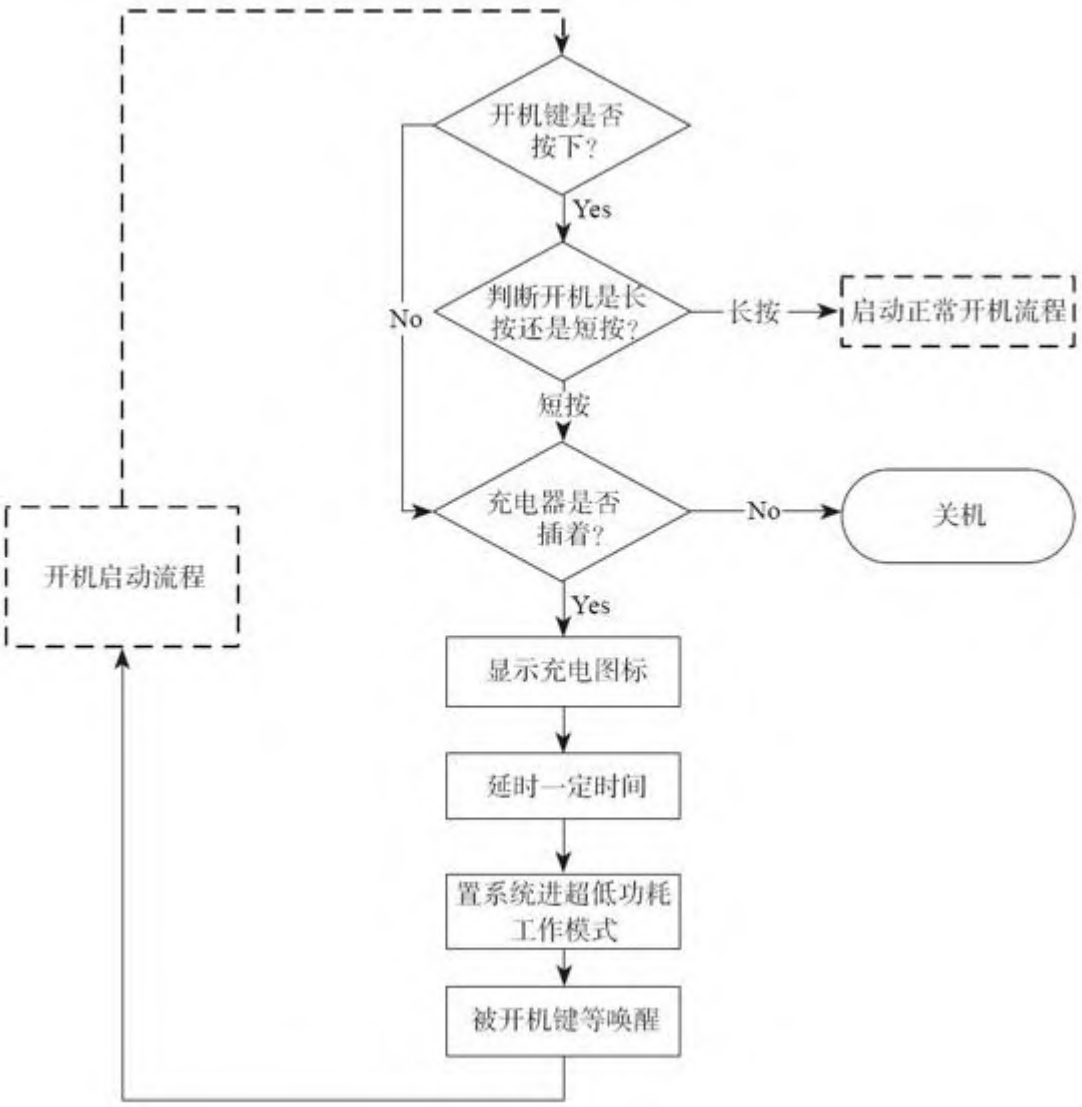


图20-3　充电开机处理流程

读累了记得休息一会哦~

公众号：古德猫宁李

- 电子书搜索下载
- 书单分享
- 书友学习交流

网站：沉金书屋 https://www.chenjin5.com

- 电子书搜索下载
- 电子书打包资源分享
- 学习资源分享

参考文献

[1] Daniel P.Bovet, Marco Cesati.Understanding the Linux Kernel[M].3rd ed.O'Reilly Media, 2005.

[2] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.Linux Device Drivers[M].3rd ed.O'Reilly Media, 2005.

[3] 宋宝华.Linux设备驱动开发详解[M].北京：人民邮电出版社, 2010.

[4] Get Started[J/OL].http://developer.android.com/about/start.html#.

[5] 陈莉君Linux内核之旅 [J/OL].http://kerneltravel.eefocus.com/.

[6] Android Low-Level System Architecture[J/OL].http://source.android.com/devices/index.html.

[7] Android display架构分析 [J/OL].http://blog.csdn.net/bonderwu/article/details/5805961.

[8] Android用户输入系统结构和移植内容 [J/OL].http://bbs.hiapk.com/thread-3196702-1-1.html.

[9] Linux内核input子系统解析 [J/OL].http://blog.csdn.net/hongtao_liu/article/details/5679171.

[10] Android系统移植 [J/OL].http://wenku.baidu.com/view/194f368183d049649b665866.html.

[11] Binder——Android的IPC通信机制[J/OL].http://blog.csdn.net/jmq_0000/article/details/7349844.

[12] Android系统vold透析[J/OL].http://wenku.baidu.com/view/20b1a4222f60ddccda38a0e7.html.

[13] Android系统启动流程——bootloader[J/OL].http://blog.csdn.net/lizhiguo0532/article/details/7017503.